

# Arm® A32/T32 Instruction Set

## for A-profile architecture



# Arm A32/T32 Instruction Set for A-profile architecture

Copyright © 2010-2024 Arm Limited (or its affiliates). All rights reserved.

## Release Information

For information on the change history and known issues for this release, see the **Release Notes** in the **A32/T32 ISA XML for A-profile architecture (2024-12)**.

## Proprietary Notice

This document is protected by copyright and other related rights and the practice 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. **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, patents, copyrights, trade secrets, 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 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 License shall be governed by English Law.

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

Copyright © 2010-2024 Arm Limited (or its affiliates). All rights reserved.

Arm Limited. Company 02557590 registered in England.  
110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-20349  
8 March 2024

## Confidentiality Status

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

## Product Status

The information relating to the 2024 Extensions is at Alpha quality. Alpha quality means that most major features of the specification are described in this release, but some features and details might be missing.

The information relating to the rest of the A-profile Architecture is at Beta quality. Beta quality means that all major features of the specification are described, but some details might be missing.

## Web Address

<http://www.arm.com>

## Progressive Terminology Commitment

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

Previous issues of this document included terms that can be offensive. We have replaced these terms. If you find offensive terms in this document, please contact [terms@arm.com](mailto:terms@arm.com).

## Feedback on this document

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

As part of the ticket, include:

- The title, *Arm® A32/T32 Instruction Set for A-profile architecture*.
- The number, DDI 0597.
- The section name to which your comments refer.
- The page number(s) to which your comments refer.
- The rule identifier(s) to which your comments refer if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.



## 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).

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

[SLS \(register\)](#): 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).

[SRS \(immediate\)](#): 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).

[SMLSD, SMLSDX](#): Signed Multiply Subtract Dual.

[SMLSLD, SMLSLDX](#): 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, SRSDB, 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).

[STMDB, 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant imm32 = 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							

## 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

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the ADC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the ADCS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    constant (result, nzcvc) = AddWithCarry(R[n], imm32, PSTATE.C);  
    if d == 15 then // Can only occur for A32 encoding  
        if setflags then  
            ALUExceptionReturn(result);  
        else  
            ALUWritePC(result);  
    else  
        R[d] = result;  
        if setflags then  
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.





## 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																															

#### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	

## Encoding for the T1 variant

```
ADC<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)
```

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

## Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);  
constant setflags = !InITBlock();  
constant (shift_t, shift_n) = (SRTYPE_LSL, 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			Rd			imm2			stype		Rm				

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

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

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

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: 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 encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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.

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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], shifted, PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;

```

### Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

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																															

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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																																

### 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					

### 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	1	0	imm3				Rd					imm8							

### Encoding for the T3 variant

ADDW{<c>}{<q>} <Rd>, PC, #<imm12> // (<Rd>, <imm12> can be represented in T1)

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

is equivalent to

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

## Assembler Symbols

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

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

<Rd>	<p>For encoding A1: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</p> <p>For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.</p>
<label>	<p>For encoding A1: the label of an instruction or literal data item whose address is to be loaded into &lt;Rd&gt;. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.</p> <p>If the offset is zero or positive, encoding A1 is used, with imm32 equal to the offset.</p> <p>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.</p> <p>Permitted values of the size of the offset are any of the constants described in <a href="#">Modified immediate constants in A32 instructions</a>.</p> <p>For encoding T1: the label of an instruction or literal data item whose address is to be loaded into &lt;Rd&gt;. 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.</p> <p>For encoding T3: the label of an instruction or literal data item whose address is to be loaded into &lt;Rd&gt;. The assembler calculates the required value of the offset from the Align(PC, 4) value of the ADR instruction to this label.</p> <p>If the offset is zero or positive, encoding T3 is used, with imm32 equal to the offset.</p> <p>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.</p> <p>Permitted values of the size of the offset are 0-4095.</p>
<imm8>	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
<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 <a href="#">Modified immediate constants in A32 instructions</a> for the range of values.

## Operation

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

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### 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";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1');
constant imm32 = 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		

#### Encoding for the T1 variant

```
ADD<c>{<q>}{<Rd>, <Rn>, #<imm3> // (Inside IT block)
```

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

#### Decode for this encoding

```
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = !InITBlock();
constant imm32 = ZeroExtend(imm3, 32);
```



## T2

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

### Encoding for the T2 variant

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

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

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant setflags = !InITBlock();
constant imm32 = ZeroExtend(imm8, 32);
```

## 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									

Rn

### Encoding for the ADD variant

Applies when (S == 0)

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

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

### Encoding for the ADDS variant

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

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

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

### Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMN (immediate)";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant imm32 = T32ExpandImm(i:imm3:imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
```

## 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									

Rn

## Encoding for the T4 variant

```
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";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = FALSE;
constant imm32 = ZeroExtend(i:imm3:imm8, 32);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the ADDS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;. Arm deprecates use of this instruction.</li></ul> For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1 and T4: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">ADD (SP plus immediate)</a> . If the PC is used, see <a href="#">ADR</a> . For encoding T1: is the general-purpose source register, encoded in the "Rn" field. For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">ADD (SP plus immediate)</a> .
<imm3>	Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T3: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

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();
    constant (result, nzcvc) = AddWithCarry(R[n], imm32, '0');
    if d == 15 then // Can only occur for A32 encoding
      if setflags then
        ALUExceptionReturn(result);
      else
        ALUWritePC(result);
    else
      R[d] = result;
      if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
  else
    if ConditionPassed\(\) then
      EncodingSpecificOperations();
      constant (result, nzcvc) = AddWithCarry(R[n], imm32, '0');
      R[d] = result;
      if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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		

Encoding for the T1 variant

```
ADD<c>{<q>} <Rd>, <Rn>, <Rm> // (Inside IT block)

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

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 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		
Rm															

Encoding for the T2 variant

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

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

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

Decode for this encoding

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)";
constant d = UInt(DN:Rdn);  constant n = d;  constant m = UInt(Rm); constant setflags = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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				
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>.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)
```

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

## 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 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdn>	<p>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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</p> <p>The assembler language allows &lt;Rdn&gt; to be specified once or twice in the assembler syntax. When used inside an IT block, and &lt;Rdn&gt; and &lt;Rm&gt; are in the range R0 to R7, &lt;Rdn&gt; must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when &lt;Rdn&gt; is specified once or twice.</p>
<Rd>	<p>For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as &lt;Rn&gt;. If the PC is used:</p> <ul style="list-style-type: none"><li>• For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>• For the ADDS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;. Arm deprecates use of this instruction.</li></ul> <p>For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.</p> <p>When used inside an IT block, &lt;Rd&gt; must be specified. When used outside an IT block, &lt;Rd&gt; is optional, and:</p> <ul style="list-style-type: none"><li>• If omitted, this register is the same as &lt;Rn&gt;.</li><li>• If present, encoding T1 is preferred to encoding T2.</li></ul>

For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: 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 encoding T1: is the first general-purpose source register, encoded in the "Rn" field.

For encoding T3: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see [ADD \(SP plus register\)](#).

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T3: is the second general-purpose source register, encoded in the "Rm" field.

For encoding T2: 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":

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

<amount> For encoding A1: 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 encoding T3: 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.

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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	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	Rn				Rd				Rs				0	stype	1	Rm				
cond																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```



## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

### 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

```
constant d = UInt(Rd); constant setflags = (S == '1'); constant imm32 = 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						

### Encoding for the T1 variant

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

### Decode for this encoding

```
constant d = UInt(Rd); constant setflags = FALSE; constant imm32 = ZeroExtend(imm8:'00', 32);
```

### 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						

## Encoding for the T2 variant

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

## Decode for this encoding

```
constant d = 13; constant setflags = FALSE; constant imm32 = ZeroExtend(imm7:'00', 32);
```

## 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							

## Encoding for the ADD variant

Applies when (S == 0)

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

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

## 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)";
constant d = UInt(Rd); constant setflags = (S == '1'); constant imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

## 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							

## Encoding for the T4 variant

```
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

```
constant d = UInt(Rd); constant setflags = FALSE; constant imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
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.

<Rd>	<p>For encoding A1: 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:</p> <ul style="list-style-type: none"> <li>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>.</li> <li>For the ADDS variant, the instruction performs an exception return, that restores <i>PSTATE</i> from <i>SPSR_&lt;current_mode&gt;</i>.</li> </ul> <p>For encoding T1: is the general-purpose destination register, encoded in the "Rd" field.</p> <p>For encoding T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.</p>
<imm8>	Is an unsigned immediate, a multiple of 4, in the range 0 to 1020, encoded in the "imm8" field as <imm8>/4.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	<p>For encoding A1: an immediate value. See <i>Modified immediate constants in A32 instructions</i> for the range of values.</p> <p>For encoding T3: an immediate value. See <i>Modified immediate constants in T32 instructions</i> for the range of values.</p>

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(R[13], imm32, '0');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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

```
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = (S == '1');  
constant (shift_t, shift_n) = 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		

Encoding for the T1 variant

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

Decode for this encoding

```
constant d = UInt(DM:Rdm);  constant m = UInt(DM:Rdm);  constant setflags = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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
Rm															

Encoding for the T2 variant

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

Decode for this encoding

```
if Rm == '1101' then SEE "encoding T1";
constant d = 13;  constant m = UInt(Rm);  constant setflags = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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			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>}.W {<Rd>}, SP, <Rm> // (<Rd>, <Rm> can be represented in T1 or T2)
```

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

## 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 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
SP,	Is the stack pointer.
<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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the ADDS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
<Rm>	For encoding A1 and T2: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T3: 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 encoding A1: 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 encoding T3: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[13], shifted, '0');
    if d == 15 then
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																																

Encoding for the A1 variant

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

Decode for this encoding

```
constant d = UInt(Rd);  constant imm32 = A32ExpandImm(imm12);  constant add = 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																																

Encoding for the A2 variant

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

Decode for this encoding

```
constant d = UInt(Rd);  constant imm32 = A32ExpandImm(imm12);  constant add = 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					

Encoding for the T1 variant

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

Decode for this encoding

```
constant d = UInt(Rd);  constant imm32 = ZeroExtend(imm8:'00', 32);  constant add = 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							

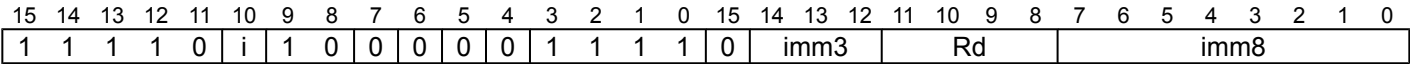
Encoding for the T2 variant

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

Decode for this encoding

```
constant d = UInt(Rd);  constant imm32 = ZeroExtend(i:imm3:imm8, 32);  constant add = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

T3



Encoding for the T3 variant

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

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

Decode for this encoding

```
constant d = UInt(Rd);  constant imm32 = ZeroExtend(i:imm3:imm8, 32);  constant add = TRUE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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 encoding A1 and A2: 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 encoding T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- <label> For encoding A1 and A2: 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 encoding T1: 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 encoding T2 and T3: 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
<a href="#">ADD (immediate, to PC)</a>		Never

Alias	Of variant	Is preferred when
<a href="#">SUB (immediate, from PC)</a>	T2	<code>i:imm3:imm8 == '000000000000'</code>
<a href="#">SUB (immediate, from PC)</a>	A2	<code>imm12 == '000000000000'</code>

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant result = if add then (Align(PC32,4) + imm32) else (Align(PC32,4) - imm32);
    if d == 15 then           // Can only occur for A32 encodings
        ALUWritePC(result);
    else
        R[d] = result;

```

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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant a32 = TRUE;
constant bits(12) imm = 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							

## 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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1'); constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the AND variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the ANDS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from <code>SPSR_&lt;current_mode&gt;</code>.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = R[n] AND imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	

## Encoding for the T1 variant

```
AND<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)

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

## Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 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				

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

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

## 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 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.



<Rd> For encoding A1: 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 encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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.

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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

### Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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								S								styp															

#### 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> // (Inside IT block)`

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	0	1	0	0	1	1	1	(0)	imm3				Rd				imm2		1	0	Rm			
S																styp																

#### Encoding for the MOV, shift or rotate by value variant

`ASR<c>.W {<Rd>}, <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)`

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a> .  For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.  For encoding T3: 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.

## 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				S								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> // (Inside IT block)

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

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

is equivalent to

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

Operation

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

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											S					styp															

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> // (Outside IT block)

is equivalent to

MOVS{<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	0	1	0	1	1	1	1	(0)	imm3			Rd			imm2			1	0	Rm			
S																styp															



Encoding for the MOVS, shift or rotate by value variant

```
ASRS.W {<Rd>, } <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
ASRS{<c>}{<q>} {<Rd>, } <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, ASR #<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 encoding A1: 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 encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32. For encoding T3: 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.

## 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				S								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> // (Outside IT block)

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			
stype S																															

#### Encoding for the Flag setting variant

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

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

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

Operation

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

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																															

Encoding for the A1 variant

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

Decode for this encoding

```
constant imm32 = SignExtend(imm24:'00', 32);
```

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 for the T1 variant

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

Decode for this encoding

```
if cond == '1110' then SEE "UDF";  
if cond == '1111' then SEE "SVC";  
constant imm32 = SignExtend(imm8:'0', 32);  
if InITBlock() then UNPREDICTABLE;
```

T2

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

Encoding for the T2 variant

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

Decode for this encoding

```
constant imm32 = SignExtend(imm11:'0', 32);  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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 for the T3 variant

```
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";
constant imm32 = SignExtend(S:J2:J1:imm6:imm11:'0', 32);
if InITBlock() then UNPREDICTABLE;
```

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 for the T4 variant

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

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

Decode for this encoding

```
constant I1 = NOT(J1 EOR S); constant I2 = NOT(J2 EOR S);
constant imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

For more information about the CONSTRAINED UNPREDICTABLE behavior, see *Architectural Constraints on UNPREDICTABLE behaviors*.  
Related encodings: *Branches and miscellaneous control*.

Assembler Symbols

- <c> For encoding A1, T2 and T4: see *Standard assembler syntax fields*.  
For encoding T1: see *Standard assembler syntax fields*. Must not be AL or omitted.  
For encoding T3: see *Standard assembler syntax fields*. <c> must not be AL or omitted.
- <q> See *Standard assembler syntax fields*.
- <label> For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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 encoding T4: 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);
```



# 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 1						
cond																															

### Encoding for the A1 variant

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

### Decode for this encoding

```
constant d = UInt(Rd);
constant integer msbit = UInt(msb);
constant integer lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE;
```

### 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	1	0	imm3				Rd				imm2		(0)	msb			

### Encoding for the T1 variant

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

### Decode for this encoding

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

### 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 <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.
<lsb>	For encoding A1: is the least significant bit to be cleared, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: 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('0', (msbit-lsbit)+1);
    // Other bits of R[d] are unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



# 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 for the A1 variant

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

### Decode for this encoding

```
if Rn == '1111' then SEE "BFC";
constant d = UInt(Rd); constant n = UInt(Rn);
constant integer msbit = UInt(msb);
constant integer lsbit = UInt(lsb);
if d == 15 then UNPREDICTABLE;
if msbit < lsbit then UNPREDICTABLE;
```

### 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			
Rn																															

### Encoding for the T1 variant

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

### Decode for this encoding

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

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: is the least significant destination bit, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: 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
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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

```
constant d = UInt(Rd); constant n = UInt(Rn);  constant setflags = (S == '1');
constant a32 = TRUE;
constant bits(12) imm = 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							

## 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

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the BIC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the BICS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = R[n] AND NOT(imm32);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	

## Encoding for the T1 variant

```
BIC<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)
```

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

## Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);  
constant setflags = !InITBlock();  
constant (shift_t, shift_n) = (SRTYPE_LSL, 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			

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

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

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

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.

<Rd> For encoding A1: 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>.

For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND NOT(shifted);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

### Operational information

- If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# 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

31	30	29	28	27	26	25	24	23	22	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				Rs				0	stype	1	Rm				
cond																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

```
BKPT{<q>} {#}<imm>
```

### Decode for this encoding

```
constant imm16 = imm12:imm4;
if cond != '1110' then UNPREDICTABLE; // 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 for the T1 variant

```
BKPT{<q>} {#}<imm>
```

### Decode for this encoding

```
constant imm16 = ZeroExtend(imm8, 16);
```

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](#). A BKPT instruction must be unconditional.
- <imm> For encoding A1: 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 encoding T1: 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);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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
!= 1111				1	0	1	1	imm24																							
cond																															

#### Encoding for the A1 variant

BL{<c>}{<q>} <label>

#### Decode for this encoding

```
constant imm32 = SignExtend(imm24:'00', 32);  constant targetInstrSet = 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																											
cond																																			

#### Encoding for the A2 variant

BLX{<c>}{<q>} <label>

#### Decode for this encoding

```
constant imm32 = SignExtend(imm24:H:'0', 32);  constant targetInstrSet = 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 for the T1 variant

BL{<c>}{<q>} <label>

#### Decode for this encoding

```
constant I1 = NOT(J1 EOR S);  constant I2 = NOT(J2 EOR S);  
constant imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);  
constant targetInstrSet = InstrSet_T32;  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### 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 for the T2 variant

BLX{<c>}{<q>}<label>

## Decode for this encoding

```
if H == '1' then UNDEFINED;
constant I1 = NOT(J1 EOR S); constant I2 = NOT(J2 EOR S);
constant imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);
constant targetInstrSet = InstrSet\_A32;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	For encoding A1, T1 and T2: see <a href="#">Standard assembler syntax fields</a> . For encoding A2: see <a href="#">Standard assembler syntax fields</a> . <c> must be AL or omitted.
<q>	See <a href="#">Standard assembler syntax fields</a> .
<label>	For encoding A1: 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 encoding A2: 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 encoding T1: 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 encoding T2: 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';
bits(32) targetAddress;
if targetInstrSet == InstrSet\_A32 then
  targetAddress = Align(PC32, 4) + imm32;
else
  targetAddress = PC32 + imm32;
SelectInstrSet(targetInstrSet);
BranchWritePC(targetAddress, BranchType\_DIRCALL);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

BLX{<c>}{<q>} <Rm>

### Decode for this encoding

```
constant m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

### 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)

### Encoding for the T1 variant

BLX{<c>}{<q>} <Rm>

### Decode for this encoding

```
constant m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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();
    constant target = R[m];
    bits(32) next_instr_addr;
    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';
    BXWritePC(target, BranchType_INDCALL);
```





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																															

Encoding for the A1 variant

BX{<c>}{<q>} <Rm>

Decode for this encoding

```
constant m = 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)

Encoding for the T1 variant

BX{<c>}{<q>} <Rm>

Decode for this encoding

```
constant m = UInt (Rm) ;  
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose register holding the address to be branched to, encoded in the "Rm" field. The PC can be used.  
For encoding T1: 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);
```



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																															

Encoding for the A1 variant

BXJ{<c>}{<q>} <Rm>

Decode for this encoding

```
constant m = UInt(Rm);
if m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	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 for the T1 variant

BXJ{<c>}{<q>} <Rm>

Decode for this encoding

```
constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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);
```

# 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

```
constant n = UInt(Rn); constant imm32 = ZeroExtend(i:imm5:'0', 32);
constant nonzero = (op == '1');
if InITBlock() then UNPREDICTABLE;
```

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);
```

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
!= 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	1	0
cond																															

Encoding for the A1 variant

CLRBHB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP();
```

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

Encoding for the T1 variant

CLRBHB{<c>}{<q>}

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_CLRBHB) then ExecuteAsNOP();
```

Assembler Symbols

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

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    Hint\_CLRBHB();
```

# 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)

### Encoding for the A1 variant

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)

### Encoding for the T1 variant

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 encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q>See [Standard assembler syntax fields](#).

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    ClearExclusiveLocal(ProcessorID());
```

# 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																															

### Encoding for the A1 variant

CLZ{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

CLZ{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UNPREDICTABLE;
```

## 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 encoding A1: is the general-purpose source register, encoded in the "Rm" field.  
For encoding T1: 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();
    constant result = CountLeadingZeroBits(R[m]);
    R[d] = result<31:0>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

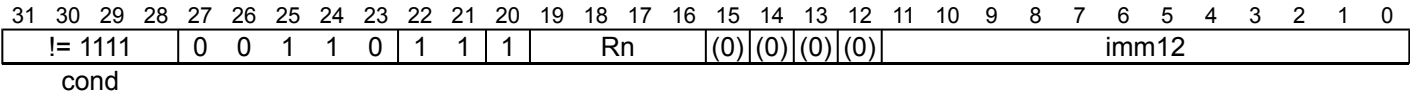
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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



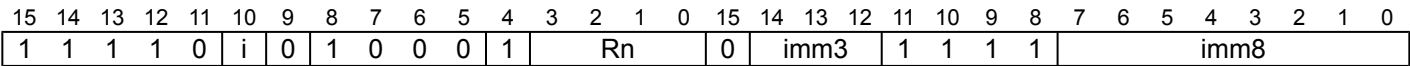
### Encoding for the A1 variant

CMN{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn);  constant imm32 = A32ExpandImm(imm12);
```

## T1



### Encoding for the T1 variant

CMN{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn);  constant imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
- <const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.  
For encoding T1: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(R[n], imm32, '0');
    PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### 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

```
constant n = UInt(Rn); constant m = UInt(Rm);  
constant (shift_t, shift_n) = 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	

#### Encoding for the T1 variant

```
CMN{<c>}{<q>} <Rn>, <Rm>
```

#### Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);  
constant (shift_t, shift_n) = (SRTType_LSL, 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			

#### 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>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

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

Decode for all variants of this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcv) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcv;
```

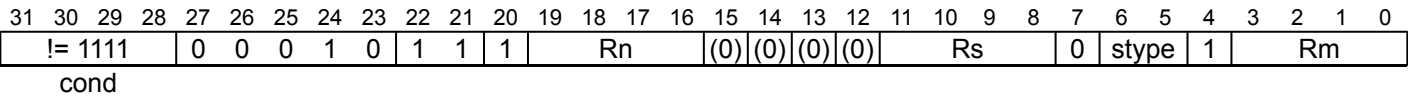
Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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 for the A1 variant

CMN{<c>}{<q>}<Rn>, <Rm>, <type> <Rs>

Decode for this encoding

```
constant n = UInt(Rn);  constant m = UInt(Rm);  constant s = UInt(Rs);
constant shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], shifted, '0');
    PSTATE.<N,Z,C,V> = nzcvc;
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# 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																															

### Encoding for the A1 variant

CMP{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn);  constant imm32 = 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						

### Encoding for the T1 variant

CMP{<c>}{<q>} <Rn>, #<imm8>

### Decode for this encoding

```
constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm8, 32);
```

## 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							

### Encoding for the T2 variant

CMP{<c>}.W <Rn>, #<const> // (<Rd>, <const> can be represented in T1)

CMP{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn);  constant imm32 = T32ExpandImm(i:imm3:imm8);
if n == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1: is a general-purpose source register, encoded in the "Rn" field.

For encoding T2: is the general-purpose source register, encoded in the "Rn" field.

<imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For encoding T2: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(imm32), '1');
    PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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

```
constant n = UInt(Rn); constant m = UInt(Rm);  
constant (shift_t, shift_n) = 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	

#### Encoding for the T1 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> both from R0-R7)
```

#### Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);  
constant (shift_t, shift_n) = (SRTType_LSL, 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		

#### Encoding for the T2 variant

```
CMP{<c>}{<q>} <Rn>, <Rm> // (<Rn> and <Rm> not both from R0-R7)
```

#### Decode for this encoding

```
constant n = UInt(N:Rn); constant m = UInt(Rm);  
constant (shift_t, shift_n) = (SRTType_LSL, 0);  
if n < 8 && m < 8 then UNPREDICTABLE;  
if n == 15 || m == 15 then UNPREDICTABLE;
```

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	1	imm2		stype		Rm			

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>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1 or T2)
CMP{<c>}{<q>} <Rn>, <Rm>, <shift> #<amount>
```

Decode for all variants of this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1 and T3: is the first general-purpose source register, encoded in the "Rn" field.  
For encoding T2: is the first general-purpose source register, encoded in the "N:Rn" field.
- <Rm>

For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1, T2 and T3: 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 encoding A1: 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 encoding T3: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

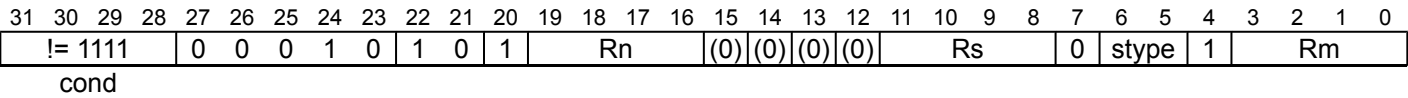
Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 for the A1 variant

```
CMP{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>
```

Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), '1');
    PSTATE.<N,Z,C,V> = nzcvc;
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# 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				

### 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 UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
constant enable = (imod == '10'); constant disable = (imod == '11');
constant changemode = (M == '1'); constant pemode = mode;
constant affectA = (A == '1'); constant affectI = (I == '1'); constant affectF = (F == '1');
if (imod == '00' && M == '0') || imod == '01' then UNPREDICTABLE;
```

## 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 != '0000' && 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

### 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 UNPREDICTABLE;
constant enable = (im == '0'); constant disable = (im == '1'); constant changemode = FALSE;
constant affectA = (A == '1'); constant affectI = (I == '1'); constant affectF = (F == '1');
constant bits(5) pemode = bits(5) UNKNOWN;
if InITBlock() then UNPREDICTABLE;
```

## 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";
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
constant enable = (imod == '10'); constant disable = (imod == '11');
constant changemode = (M == '1'); constant pemode = mode;
constant affectA = (A == '1'); constant affectI = (I == '1'); constant affectF = (F == '1');
if imod == '01' || InITBlock() then UNPREDICTABLE;
```

### 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 00 and the M bit is 0, 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 <a href="#">Standard assembler syntax fields</a> .
<iflags>	Is a sequence of one or more of the following, specifying which interrupt mask bits are affected: <ul style="list-style-type: none"><li><b>a</b> Sets the A bit in the instruction, causing the specified effect on <a href="#">PSTATE.A</a>, the SEError interrupt mask bit.</li><li><b>i</b> Sets the I bit in the instruction, causing the specified effect on <a href="#">PSTATE.I</a>, the IRQ interrupt mask bit.</li><li><b>f</b> Sets the F bit in the instruction, causing the specified effect on <a href="#">PSTATE.F</a>, the FIQ interrupt mask bit.</li></ul>
<mode>	Is the number of the mode to change to, in the range 0 to 31, encoded in the "mode" field.

## Operation

```
if CurrentInstrSet() == InstrSet A32 then
    EncodingSpecificOperations();
    if PSTATE.EL != ELO then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(pemode);
else
    EncodingSpecificOperations();
    if PSTATE.EL != ELO then
        if enable then
            if affectA then PSTATE.A = '0';
            if affectI then PSTATE.I = '0';
            if affectF then PSTATE.F = '0';
        if disable then
            if affectA then PSTATE.A = '1';
            if affectI then PSTATE.I = '1';
            if affectF then PSTATE.F = '1';
        if changemode then
            // AArch32.WriteModeByInstr() sets PSTATE.IL to 1 if this is an illegal mode change.
            AArch32.WriteModeByInstr(pemode);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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;
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant integer size = 8 << UInt(sz);
constant crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

## 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;.

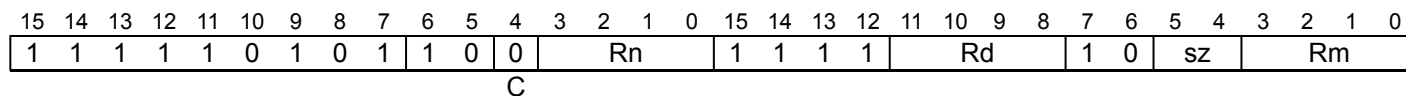
If `cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The instruction executes unconditionally.
- The instruction executes conditionally.

## T1

### (FEAT\_CRC32)



### 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_CRC32) then UNDEFINED;
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant integer size = 8 << UInt(sz);
constant crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> . 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();

    constant acc = R[n];           // accumulator
    constant val = R[m]<size-1:0>; // input value
    constant poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    constant tempacc = BitReverse(acc):Zeros(size);
    constant tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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;
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant integer size = 8 << UInt(sz);
constant crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
if cond != '1110' then UNPREDICTABLE;
```

## CONSTRAINED UNPREDICTABLE behavior

If size == 64, then one of the following behaviors must occur:

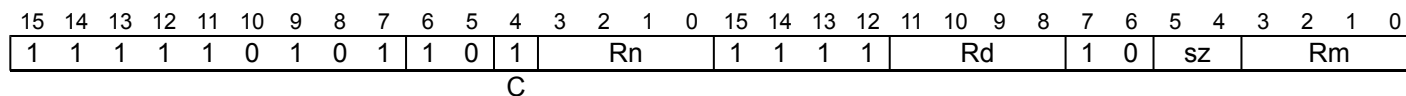
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: size = 32;.

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)



### 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_CRC32) then UNDEFINED;
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant integer size = 8 << UInt(sz);
constant crc32c = (C == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if size == 64 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> . 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();

    constant acc = R[n];           // accumulator
    constant val = R[m]<size-1:0>;   // input value
    constant poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    constant tempacc = BitReverse(acc):Zeros(size);
    constant tempval = BitReverse(val):Zeros(32);
    // Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation
    R[d] = BitReverse(Poly32Mod2(tempacc EOR tempval, poly));
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	0	0
cond																															

### Encoding for the A1 variant

CSDB{<c>}{<q>}

### Decode for this encoding

```
if cond != '1110' then UNPREDICTABLE; // 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

### Encoding for the T1 variant

CSDB{<c>}{<q>}

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

### 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\(\);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

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			

### Encoding for the T1 variant

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();
```

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

Encoding for the T1 variant

DCPS1

Decode for this encoding

```
// No additional decoding required.
```

## Operation

```
if !Halted() then UNDEFINED;

if EL2Enabled() && PSTATE.EL == EL0 then
    constant tge = if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    if tge == '1' then UNDEFINED;

if PSTATE.EL != EL0 || ELUsingAArch32(EL1) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    if PSTATE.EL != EL2 then
        AArch32.WriteMode(M32_Svc);
        PSTATE.E = SCTLR.EE;
        if IsFeatureImplemented(FEAT_PAN) && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
        LR_svc = bits(32) UNKNOWN;
        SPSR_svc = bits(32) UNKNOWN;
    else
        PSTATE.E = HSCTLR.EE;
        ELR_hyp = bits(32) UNKNOWN;
        HSR = bits(32) UNKNOWN;
        SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else // Targeting EL1 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL1);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL1;
    if IsFeatureImplemented(FEAT_PAN) && SCTLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';

    ELR_EL1 = bits(64) UNKNOWN;
    ESR_EL1 = bits(64) UNKNOWN;
    SPSR_EL1 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    // SCTLR_EL1.IESB might be ignored in Debug state.
    if (IsFeatureImplemented(FEAT_IESB) && SCTLR_EL1.IESB == '1' &&
        !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
        SynchronizeErrors();

UpdateEDSCRFIELDS(); // Update EDSCR PE state flags
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

Encoding for the T1 variant

DCPS2

Decode for this encoding

```
if !HaveEL(EL2) then UNDEFINED;
```

## Operation

```
if !Halted() || !EL2Enabled() then UNDEFINED;

if ELUsingAArch32(EL2) then
    AArch32.WriteMode(M32_Hyp);
    PSTATE.E = HSCTLR.EE;

    ELR_hyp = bits(32) UNKNOWN;
    HSR = bits(32) UNKNOWN;
    SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
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';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';

    ELR_EL2 = bits(64) UNKNOWN;
    ESR_EL2 = bits(64) UNKNOWN;
    SPSR_EL2 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    // SCTLR_EL2.IESB might be ignored in Debug state.
    if (IsFeatureImplemented(FEAT_IESB) && SCTLR_EL2.IESB == '1' &&
        !ConstrainUnpredictableBool(Unpredictable_IESBinDebug)) then
        SynchronizeErrors();

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

Encoding for the T1 variant

DCPS3

Decode for this encoding

```
if !HaveEL(EL3) then UNDEFINED;
```

## Operation

```
if !Halted() || EDSCR.SDD == '1' then UNDEFINED;

if ELUsingAArch32(EL3) then
    constant from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLR.SPAN == '0' then
            PSTATE.PAN = '1';
        PSTATE.E = SCTLR.EE;

    LR_mon = bits(32) UNKNOWN;
    SPSPR_mon = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
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';

    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(64) UNKNOWN;
    SPSPR_EL3 = bits(64) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLR_EL3.IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && EffectiveEA() == '1' && SCR_EL3.NMEA == '1' then
        sync_errors = TRUE;
    // SCTLR_EL3.IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then SynchronizeErrors();

UpdateEDSCRFields(); // Update EDSCR PE state flags
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

Encoding for the A1 variant

```
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			

Encoding for the T1 variant

```
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 encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.  
For encoding T1: 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.

**ISHLD**

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.

**NSHLD**

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.

**OSHLD**

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();
    MBReqDomain domain;
    MBReqTypes types;
    case option of
        when '0001' domain = MBReqDomain OuterShareable; types = MBReqTypes Reads;
        when '0010' domain = MBReqDomain OuterShareable; types = MBReqTypes Writes;
        when '0011' domain = MBReqDomain OuterShareable; types = MBReqTypes All;
        when '0101' domain = MBReqDomain Nonshareable; types = MBReqTypes Reads;
        when '0110' domain = MBReqDomain Nonshareable; types = MBReqTypes Writes;
        when '0111' domain = MBReqDomain Nonshareable; types = MBReqTypes All;
        when '1001' domain = MBReqDomain InnerShareable; types = MBReqTypes Reads;
        when '1010' domain = MBReqDomain InnerShareable; types = MBReqTypes Writes;
        when '1011' domain = MBReqDomain InnerShareable; types = MBReqTypes All;
        when '1101' domain = MBReqDomain FullSystem; types = MBReqTypes Reads;
        when '1110' domain = MBReqDomain FullSystem; types = MBReqTypes Writes;
        otherwise domain = MBReqDomain FullSystem; types = MBReqTypes All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MBReqDomain FullSystem;
        if HCR.BSU == '10' && domain != MBReqDomain FullSystem then
            domain = MBReqDomain OuterShareable;
        if HCR.BSU == '01' && domain == MBReqDomain Nonshareable then
            domain = MBReqDomain InnerShareable;

    DataMemoryBarrier(domain, types);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			
option																															

Encoding for the A1 variant

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			
option																															

Encoding for the T1 variant

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 encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.  
For encoding T1: 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();
    boolean nXS;
    if IsFeatureImplemented(FEAT_XS) then
        nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
            IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1');
    else
        nXS = FALSE;
    MReqDomain domain;
    MReqTypes types;
    case option of
        when '0001' domain = MReqDomain OuterShareable; types = MReqTypes Reads;
        when '0010' domain = MReqDomain OuterShareable; types = MReqTypes Writes;
        when '0011' domain = MReqDomain OuterShareable; types = MReqTypes All;
        when '0101' domain = MReqDomain Nonshareable; types = MReqTypes Reads;
        when '0110' domain = MReqDomain Nonshareable; types = MReqTypes Writes;
        when '0111' domain = MReqDomain Nonshareable; types = MReqTypes All;
        when '1001' domain = MReqDomain InnerShareable; types = MReqTypes Reads;
        when '1010' domain = MReqDomain InnerShareable; types = MReqTypes Writes;
        when '1011' domain = MReqDomain InnerShareable; types = MReqTypes All;
        when '1101' domain = MReqDomain FullSystem; types = MReqTypes Reads;
        when '1110' domain = MReqDomain FullSystem; types = MReqTypes Writes;
    otherwise
        assert option != '0x00';
        domain = MReqDomain FullSystem; types = MReqTypes All;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR.BSU == '11' then
            domain = MReqDomain FullSystem;
        if HCR.BSU == '10' && domain != MReqDomain FullSystem then
            domain = MReqDomain OuterShareable;
        if HCR.BSU == '01' && domain == MReqDomain Nonshareable then
            domain = MReqDomain InnerShareable;

    DataSynchronizationBarrier(domain, types, nXS);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant a32 = TRUE;
constant bits(12) imm = 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							

## 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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1');
constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the EOR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the EORS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = R[n] EOR imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:



- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	

Encoding for the T1 variant

```
EOR<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)

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

Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 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					

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

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

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 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

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

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "TEQ (register)";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
```

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

<Rd> For encoding A1: 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 encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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.

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();
    constant (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] EOR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

### Operational information

- If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# 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																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 third general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] EOR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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)	0	1	1	0	(1)	(1)	(1)	(0)
cond																															

### Encoding for the A1 variant

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

### Encoding for the T1 variant

ERET{<c>}{<q>}

### Decode for this encoding

```
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        constant new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R\[14\];
        AArch32.ExceptionReturn(new_pc_value, SPSR\_curr[]);
else
    if PSTATE.M == M32\_User then
        UNDEFINED;
    elseif PSTATE.M == M32\_System then
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        SynchronizeContext();
        DebugRestorePSR();
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

ESB{<c>}{<q>}

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP();  
if cond != '1110' then UNPREDICTABLE; // 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

### Encoding for the T1 variant

ESB{<c>}{<q>}

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_RAS) then ExecuteAsNOP();  
if InITBlock() then UNPREDICTABLE;
```

## 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();
    elsif IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && !ELUsingAArch32(EL3) then
        AArch64.dESBOperation();
    TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

HLT{<q>} {#}<imm>

### Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;      // 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 for the T1 variant

HLT{<q>} {#}<imm>

### Decode for this encoding

```
if EDSCR.HDE == '0' || !HaltingAllowed() then UNDEFINED;
```

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 encoding A1: 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 encoding T1: 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();  
constant boolean is_async = FALSE;  
constant FaultRecord fault = NoFault();  
Halt(DebugHalt\_HaltInstruction, is_async, fault);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

HVC{<q>} {#}<imm16>

Decode for this encoding

```
if cond != '1110' then UNPREDICTABLE;
constant imm16 = 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											

Encoding for the T1 variant

HVC{<q>} {#}<imm16>

Decode for this encoding

```
constant imm16 = imm4:imm12;
if InITBlock() then UNPREDICTABLE;
```

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

## Assembler Symbols

<q>	See <a href="#">Standard assembler syntax fields</a> . An HVC instruction must be unconditional.
<imm16>	<p>For encoding A1: 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.</p> <p>For encoding T1: 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.</p>

## Operation

```
EncodingSpecificOperations();
if PSTATE.EL IN {EL0, EL3} || !EL2Enabled() then
    UNDEFINED;

bit hvc_enable;
if HaveEL(EL3) then
    if ELUsingAArch32(EL3) && SCR.HCE == '0' && PSTATE.EL == EL2 then
        UNPREDICTABLE;
    else
        hvc_enable = SCR_curr[].HCE;
else
    hvc_enable = if ELUsingAArch32(EL2) then NOT(HCR.HCD) else NOT(HCR_EL2.HCD);

if hvc_enable == '0' then
    UNDEFINED;
else
    AArch32.CallHypervisor(imm16);
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

Encoding for the A1 variant

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			

Encoding for the T1 variant

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 encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.  
For encoding T1: 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();
```

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 for the T1 variant

IT{<x>{<y>{<z>}}}{<q>}<cond>

Decode for this encoding

```
if mask == '0000' then SEE "Related encodings";
if firstcond == '1111' || (firstcond == '1110' && BitCount(mask) != 1) then UNPREDICTABLE;
if InITBlock() then UNPREDICTABLE;
```

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: <div><div>T</div><div>firstcond[0]</div><div>E</div><div>NOT firstcond[0]</div></div>
<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: <div><div>T</div><div>firstcond[0]</div></div>



**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;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	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)	0	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																															

Encoding for the A1 variant

LDA{<c>}{<q>} <Rt>, [[<Rn>](#)]

Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	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	1	0	(1)	(1)	(1)	(1)

Encoding for the T1 variant

LDA{<c>}{<q>} <Rt>, [[<Rn>](#)]

Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    R[t] = MemO[address, 4];
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# 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																															

### Encoding for the A1 variant

LDAB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDAB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    R[t] = ZeroExtend(MemQ[address, 1], 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# 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																															

### Encoding for the A1 variant

LDAEX{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDAEX{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemO(address, 4);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDAEXB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(MemO[address, 1], 32);
```



**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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	0	1	1	Rn				Rt				(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)
cond																															

#### Encoding for the A1 variant

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

#### Decode for this encoding

```
constant t = UInt(Rt); constant t2 = t + 1; constant n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

### 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)

#### Encoding for the T1 variant

LDAEXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

#### Decode for this encoding

```
constant t = UInt(Rt); constant t2 = UInt(Rt2); constant n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.  For encoding T1: 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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address, 8);
    constant value = MemO[address, 8];
    // 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>;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDAEXH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address, 2);
    R[t] = ZeroExtend(MemO[address, 2], 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDAH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDAH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    R[t] = ZeroExtend(MemQ[address, 2], 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



## 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												Rn																			

#### 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)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
constant n = UInt(Rn); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32); constant index = (P == '1');
constant add = (U == '1'); constant wback = (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							
Rn																															

#### 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)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
constant n = UInt(Rn); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32); constant index = (P == '1');
constant add = (U == '1'); constant wback = (W == '1');
```

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see <a href="#">LDC (literal)</a> .						
<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.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table border="1" data-bbox="240 1397 384 1491"> <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.						

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = 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;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



## 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																															

### Encoding for the A1 variant

Applies when  $(!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0))$

LDC{<c>}{<q>} p14, c5, <label>

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;
constant index = (P == '1'); constant add = (U == '1'); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

### CONSTRAINED UNPREDICTABLE behavior

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 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							

### Encoding for the T1 variant

Applies when  $(!(P == 0 \ \&\& \ U == 0 \ \&\& \ W == 0))$

LDC{<c>}{<q>} p14, c5, <label>

LDC{<c>}{<q>} p14, c5, [PC, #{+/-}<imm>]

### Decode for this encoding

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
constant index = (P == '1'); constant add = (U == '1'); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32);
if W == '1' || (P == '0' && CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

If `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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<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.						
<label>	<p>The label of the literal data item that is to be loaded into &lt;Rt&gt;. 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.</p> <p>If the offset is zero or positive, imm32 is equal to the offset and add == TRUE (encoded as U == 1).</p> <p>If the offset is negative, imm32 is equal to minus the offset and add == FALSE (encoded as U == 0).</p>						
+/-	<p>Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":</p> <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.						

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();
    constant offset_addr = if add then (Align(PC32,4) + imm32) else (Align(PC32,4) - imm32);
    constant address = if index then offset_addr else Align(PC32,4);

    // System register write to DBGDTRTXint.
    AArch32.SysRegWriteM(cp, ThisInstr(), address);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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.

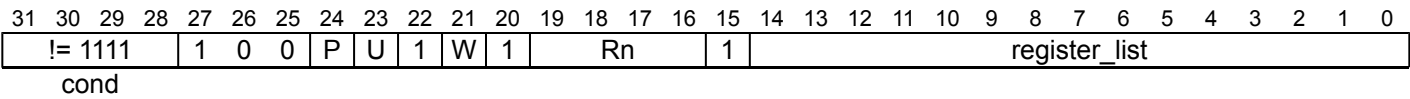
The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

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 for the A1 variant

```
LDM{<amode>}{<c>}{<q>} <Rn>{!}, <registers_with_pc>^
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list;
constant wback = (W == '1'); constant increment = (U == '1'); constant wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

### 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:
<b>DA</b>	Decrement After. The consecutive memory addresses end at the address in the base register. Encoded as P = 0, U = 0.
<b>FA</b>	Full Ascending. For this instruction, a synonym for DA.
<b>DB</b>	Decrement Before. The consecutive memory addresses end one word below the address in the base register. Encoded as P = 1, U = 0.
<b>EA</b>	Empty Ascending. For this instruction, a synonym for DB.
<b>IA</b>	Increment After. The consecutive memory addresses start at the address in the base register. This is the default. Encoded as P = 0, U = 1.
<b>FD</b>	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.

! 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
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        constant length = 4*BitCount(registers) + 4;
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;

        for i = 0 to 14
            if registers<i> == '1' then
                R[i] = MemS[address,4]; address = address + 4;
        constant new_pc_value = MemS[address,4];

        if wback && registers<n> == '0' then
            R[n] = if increment then R[n]+length else R[n]-length;
        if wback && registers<n> == '1' then
            R[n] = bits(32) UNKNOWN;

        AArch32.ExceptionReturn(new_pc_value, SPSR\_curr[]);
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

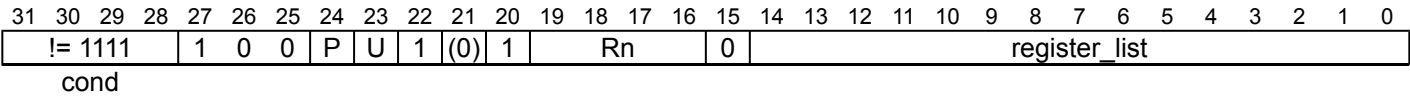
## LDM (User registers)

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 for the A1 variant

```
LDM{<amode>}{<c>}{<q>} <Rn>, <registers_without_pc>^
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant increment = (U == '1');
constant wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### 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 UNPREDICTABLE;
    else
        constant length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then // Load User mode register
                Rmode[i, M32_User] = MemS[address,4]; address = address + 4;
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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.

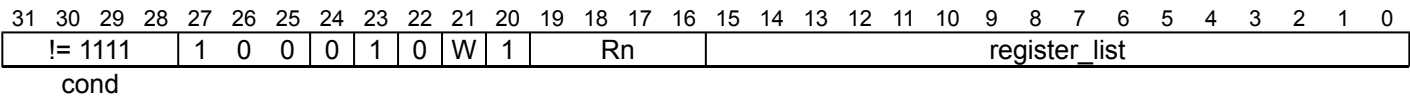
The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

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



## Encoding for the A1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

## Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

## CONSTRAINED UNPREDICTABLE behavior

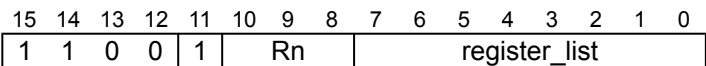
If `BitCount(registers) < 1`, then one of the following behaviors must occur:

- 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



Encoding for the T1 variant

```
LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
constant n = UInt(Rn);  constant registers = '00000000':register_list;
constant wback = (registers<n> == '0');
if BitCount(registers) < 1 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	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	1	Rn			P	M	register_list														

Encoding for the T2 variant

```
LDM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented

LDMFD{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack, if <Rn>, '!' and <registers> can be represented

LDM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

Decode for this encoding

```
constant n = UInt(Rn);  constant registers = P:M:register_list;  constant wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
!	For encoding A1 and T2: 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 encoding T1: 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 encoding A1: 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 encoding T1: 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 encoding T2: 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: <ul style="list-style-type: none"> <li>• The LR must not be in the list.</li> <li>• The instruction must be either outside any IT block, or the last instruction in an IT block.</li> </ul>

## Alias Conditions

Alias	Of variant is preferred when	
<a href="#">POP (multiple registers)</a>	T2	<code>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(P:M:register_list) &gt; 1</code>
<a href="#">POP (multiple registers)</a>	A1	<code>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(register_list) &gt; 1</code>

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4]; address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# LMDMA, 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

31	30	29	28	27	26	25	24	23	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	0	0	W	1	Rn				register_list															
cond																															

### Encoding for the A1 variant

```
LMDMA{<c>}{<q>}<Rn>{!}, <registers> // (Preferred syntax)

LDMFA{<c>}{<q>}<Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];    address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

## Encoding for the A1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

## Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

## 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													

## Encoding for the T1 variant

```
LDMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

## Decode for this encoding

```
constant n = UInt(Rn); constant registers = P:M:register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 2 || (P == '1' && M == '1') then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## 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

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: 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 encoding T1: 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();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4]; address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] - 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

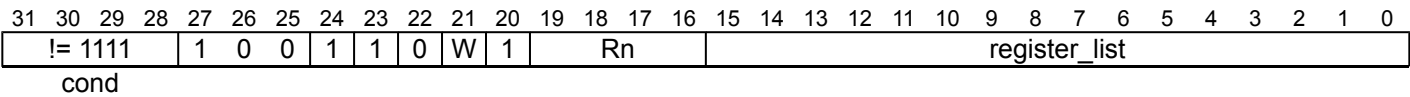
# LDMIB, LDMED

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 for the A1 variant

```
LDMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

LDMED{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
```

## 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 <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>	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();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = MemS[address,4];    address = address + 4;
    if registers<15> == '1' then
        LoadWritePC(MemS[address,4]);
    if wback && registers<n> == '0' then R[n] = R[n] + 4*BitCount(registers);
    if wback && registers<n> == '1' then R[n] = bits(32) UNKNOWN;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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)";
if P == '0' && W == '1' then SEE "LDRT";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if wback && n == t then UNPREDICTABLE;
```

### 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		

Encoding for the T1 variant

```
LDR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm5:'00', 32);
constant index = TRUE;  constant add = TRUE;    constant wback = 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 for the T2 variant

```
LDR{<c>}{<q>} <Rt>, [SP{, #<+><imm>}]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = 13;  constant imm32 = ZeroExtend(imm8:'00', 32);
constant index = TRUE;  constant add = TRUE;  constant wback = 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	1	!= 1111			Rt			imm12													
Rn																															

Encoding for the T3 variant

```
LDR{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

LDR{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE;  constant add = TRUE;
constant wback = FALSE; if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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								
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)";
if P == '1' && U == '1' && W == '0' then SEE "LDRT";
if P == '0' && W == '0' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8, 32); constant index = (P == '1'); constant add = (U == '1');
constant wback = (W == '1');
if (wback && n == t) || (t == 15 && InITBlock() && !LastInITBlock()) then UNPREDICTABLE;
```

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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Rt>	<p>For encoding A1: 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 <i>Pseudocode description of operations on the AArch32 general-purpose registers and the PC</i>.</p> <p>For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.</p> <p>For encoding T3 and T4: 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 <i>Pseudocode description of operations on the AArch32 general-purpose registers and the PC</i>.</p>
<Rn>	<p>For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see <i>LDR (literal)</i>.</p> <p>For encoding T1: is the general-purpose base register, encoded in the "Rn" field.</p>
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

- +

Specifies the offset is added to the base register.
- <imm>

For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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 encoding T4: 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.

Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">POP (single register)</a>	A1 (post-indexed)	<code>P == '0' &amp;&amp; U == '1' &amp;&amp; W == '0' &amp;&amp; Rn == '1101' &amp;&amp; imm12 == '000000000100'</code>
<a href="#">POP (single register)</a>	T4 (post-indexed)	<code>Rn == '1101' &amp;&amp; P == '0' &amp;&amp; U == '1' &amp;&amp; W == '1' &amp;&amp; imm8 == '00000100'</code>

Operation

```
if CurrentInstrSet() == InstrSet\_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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																																

### Encoding for the A1 variant

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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32);
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if wback then UNPREDICTABLE;
```

## 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 for the T1 variant

```
LDR{<c>}{<q>} <Rt>, <label> // (Normal form)
```

### Decode for this encoding

```
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm8:'00', 32); constant add = 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											



Encoding for the T2 variant

```
LDR{<c>}.W <Rt>, <label> // (Preferred syntax, and <Rt>, <label> can be represented in T1)

LDR{<c>}{<q>} <Rt>, <label> // (Preferred syntax)

LDR{<c>}{<q>} <Rt>, [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
constant t = UInt(Rt);  constant imm32 = ZeroExtend(imm12, 32);  constant add = (U == '1');
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.  
  
For encoding T2: 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 encoding A1 and T2: 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 encoding T1: 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":

U	+/-
0	-
1	+
  - <imm> For encoding A1: 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 encoding T2: 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();
    constant base = Align(PC32, 4);
    constant address = if add then (base + imm32) else (base - imm32);
    constant data = MemU[address, 4];
    if t == 15 then
        if address<1:0> == '00' then
            LoadWritePC(data);
        else
            UNPREDICTABLE;
    else
        R[t] = data;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

## 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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

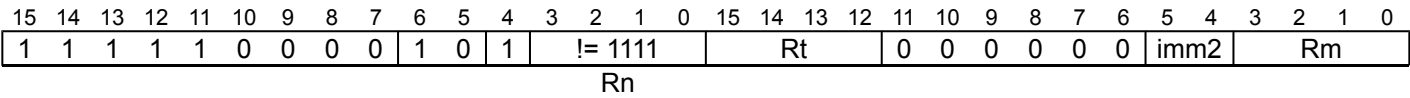
Encoding for the T1 variant

```
LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
```

T2



Encoding for the T2 variant

```
LDR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
if t == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1: is the general-purpose register to be transferred, encoded in the "Rt" field.  
For encoding T2: 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 encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.  
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
- +/-  

U	+/-
0	-
1	+

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U".
- + Specifies the index register is added to the base register.
- <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](#).

<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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,4];
    if wback then R[n] = offset_addr;
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = (R[n] + offset);
    constant address = offset_addr;
    constant data = MemU[address,4];
    if t == 15 then
      if address<1:0> == '00' then
        LoadWritePC(data);
      else
        UNPREDICTABLE;
    else
      R[t] = data;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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)";
if P == '0' && W == '1' then SEE "LDRBT";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

## 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		

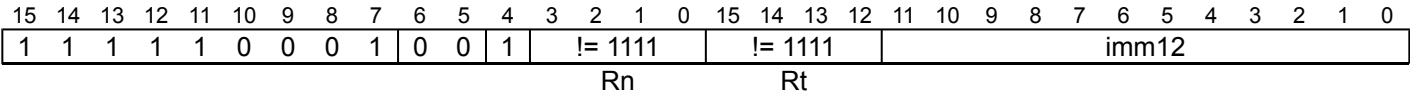
Encoding for the T1 variant

```
LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm5, 32);
constant index = TRUE;  constant add = TRUE;    constant wback = FALSE;
```

T2



Encoding for the T2 variant

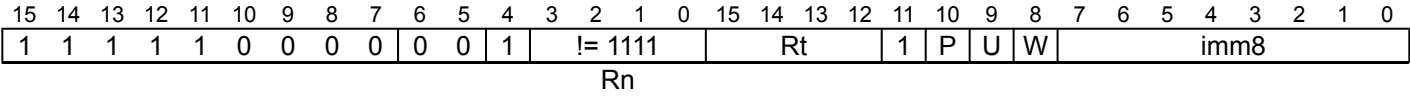
```
LDRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE;  constant add = TRUE;    constant wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3



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)";
if Rn == '1111' then SEE "LDRB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRBT";
if P == '0' && W == '0' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// 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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see <a href="#">LDRB (literal)</a> . For encoding T1: 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	+						
+	Specifies the offset is added to the base register.						
<imm>	For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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.

## Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
else
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																																

### Encoding for the A1 variant

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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32);
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

### 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	0	0	0	U	0	0	1	1	1	1	1	!= 1111					imm12											
Rt																																

### Encoding for the T1 variant

```
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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32); constant add = (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>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 encoding A1: 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 encoding T1: 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();
    constant base = Align(PC32, 4);
    constant address = if add then (base + imm32) else (base - imm32);
    R[t] = ZeroExtend(MemU[address, 1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

Encoding for the T1 variant

```
LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	0	imm2			Rm			
Rn																Rt															

Encoding for the T2 variant

```
LDRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.  
For encoding T1 and T2: 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	+
- + Specifies the index register is added to the base register.
- <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](#).
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    R[t] = ZeroExtend(MemU[address,1],32);
    if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDRBT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = FALSE; constant imm32 = ZeroExtend(imm12, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

## 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				stype		0	Rm				
cond																															

Encoding for the A2 variant

```
LDRBT{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

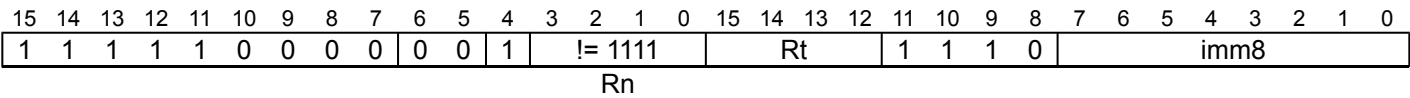
```
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = TRUE;
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
LDRBT{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRB (literal)";
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = FALSE; constant add = TRUE;
constant register_form = FALSE; constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.  
For encoding A2 and T1: 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 encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+



For encoding A2: 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> 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.
- <imm> For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    R[t] = ZeroExtend(MemU_unpriv[address, 1], 32);
    if postindex then R[n] = offset_addr;
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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 == 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)";
if Rt<0> == '1' then UNPREDICTABLE;
constant t = UInt(Rt); constant t2 = t + 1; constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm4H:imm4L, 32); constant index = (P == '1');
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

## 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:

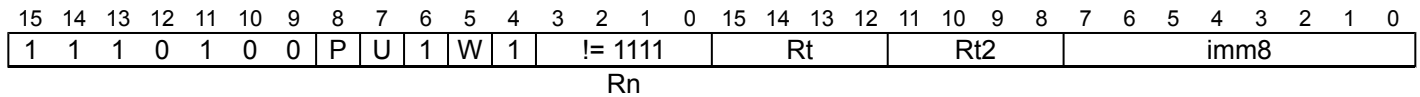
- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as an LDRD using one of offset, post-indexed, or pre-indexed addressing.

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



### 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";
if Rn == '1111' then SEE "LDRD (literal)";
constant t = UInt(Rt); constant t2 = UInt(Rt2); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
```

## 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 [Standard assembler syntax fields](#).

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

<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.						
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.  For encoding T1: 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 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 encoding A1: 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 encoding T1: 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();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    if IsAligned(address, 8) then
        constant data = MemA[address,8];
        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>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
    if wback then R[n] = offset_addr;

```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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																															

### Encoding for the A1 variant

```
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 UNPREDICTABLE;
constant t = UInt(Rt); constant t2 = t + 1;
constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant add = (U == '1');
if t2 == 15 then UNPREDICTABLE;
```

### 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							

Encoding for the T1 variant

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";
constant t = UInt(Rt); constant t2 = UInt(Rt2);
constant imm32 = ZeroExtend(imm8:'00', 32); constant add = (U == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
if W == '1' then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.  For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.
<label>	For encoding A1: 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 encoding T1: 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”:

U	+/-
0	-
1	+

<imm> For encoding A1: 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 encoding T1: 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();
    constant address = if add then (Align(PC32,4) + imm32) else (Align(PC32,4) - imm32);
    if IsAligned(address, 8) then
        constant data = MemA[address,8];
        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>;
    else
        R[t] = MemA[address,4];
        R[t2] = MemA[address+4,4];
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

### 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 UNPREDICTABLE;
constant t = UInt(Rt); constant t2 = t + 1;
constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1');
constant wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 || m == t || m == t2 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

## 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":
- | U | +/- |
|---|-----|
| 0 | -   |
| 1 | +   |
- `<Rm>` Is the general-purpose index register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    constant address = if index then offset_addr else R[n];
    if IsAligned(address, 8) then
        constant data = MemA[address, 8];
        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>;
    else
        R[t] = MemA[address, 4];
        R[t2] = MemA[address+4, 4];

    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

### Encoding for the A1 variant

LDREX{<c>}{<q>} <Rt>, [<Rn> {, {#}<imm>}]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = Zeros(32); // Zero offset
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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							

### Encoding for the T1 variant

LDREX{<c>}{<q>} <Rt>, [<Rn> {, #<imm>}]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm8:'00', 32);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. For encoding T1: 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();
    constant address = R[n] + imm32;
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = MemA[address, 4];
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDREXB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address,1);
    R[t] = ZeroExtend(MemA[address,1], 32);
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = t + 1;  constant n = UInt(Rn);
if Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
```

## 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)

### Encoding for the T1 variant

LDREXD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant n = UInt(Rn);
if t == 15 || t2 == 15 || t == t2 || n == 15 then UNPREDICTABLE;
// 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14.  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>.  For encoding T1: 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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address, 8);
    constant value = MemA[address, 8];

    // 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>;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

LDREXH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    AArch32.SetExclusiveMonitors(address,2);
    R[t] = ZeroExtend(MemA[address,2], 32);
```



**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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												Rn																			

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)";
if P == '0' && W == '1' then SEE "LDRHT";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

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		

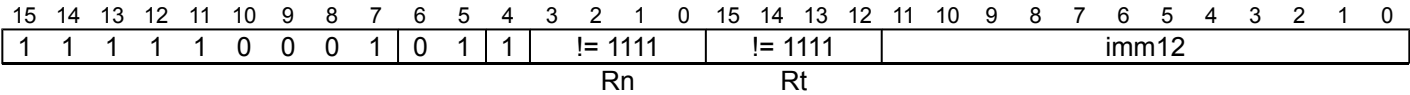
Encoding for the T1 variant

```
LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm5:'0', 32);
constant index = TRUE;  constant add = TRUE;    constant wback = FALSE;
```

T2



Encoding for the T2 variant

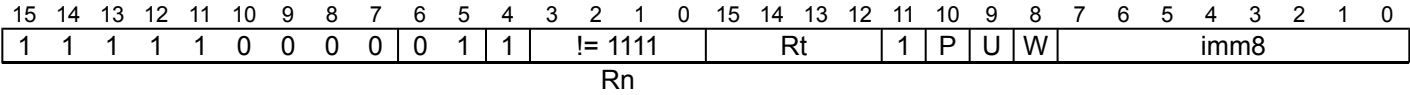
```
LDRH{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)

LDRH{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLD (immediate)";
if Rn == '1111' then SEE "LDRH (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE;  constant add = TRUE;    constant wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T3



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)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "PLDW (immediate)";
if P == '1' && U == '1' && W == '0' then SEE "LDRHT";
if P == '0' && W == '0' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// 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 <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.						
<Rn>	For encoding A1, T2 and T3: is the general-purpose base register, encoded in the "Rn" field. For PC use see <i>LDRH (literal)</i> . For encoding T1: 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	+						
+	Specifies the offset is added to the base register.						
<imm>	For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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.

## Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    constant data = MemU(address, 2);
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
else
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    constant data = MemU(address, 2);
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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	1	Rt				imm4H				1	0	1	1	imm4L			
cond																															

### Encoding for the A1 variant

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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

## 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 0 0 0								U	0 1		1	1 1 1 1				!= 1111				imm12											
Rt																															

### Encoding for the T1 variant

```
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)";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32); constant add = (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 encoding A1: 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 encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant base = Align(PC32, 4);
    constant address = if add then (base + imm32) else (base - imm32);
    constant data = MemU[address, 2];
    R[t] = ZeroExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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



Encoding for the T1 variant

```
LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	1	1	!= 1111			!= 1111			0			0	0	0	0	0	imm2			Rm		
Rn												Rt																			

Encoding for the T2 variant

```
LDRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
if Rt == '1111' then SEE "PLDW (register)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.  
For encoding T1 and T2: 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	+
- + Specifies the index register is added to the base register.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### Encoding for the A1 variant

```
LDRHT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}
```

#### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = FALSE;
constant imm32 = ZeroExtend(imm4H:imm4L, 32); constant m = integer UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### 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																															

Encoding for the A2 variant

```
LDRHT{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

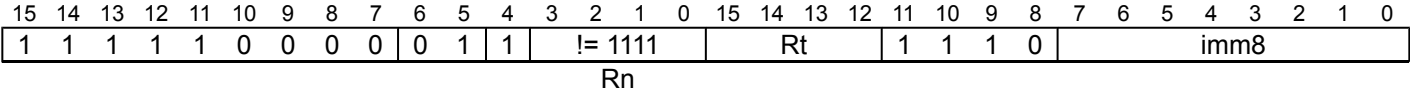
```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE;
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
LDRHT{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRH (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant postindex = FALSE;  constant add = TRUE;
constant register_form = FALSE;  constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<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 encoding A1: 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	+						

For encoding A2: 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 offset is added to the base register.
- <imm> For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then R[m] else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if postindex then R[n] else offset_addr;
    constant data = MemU_unpriv(address, 2);
    if postindex then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												Rn																			

#### 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)";
if P == '0' && W == '1' then SEE "LDRSBT";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

### 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													
Rn															Rt																

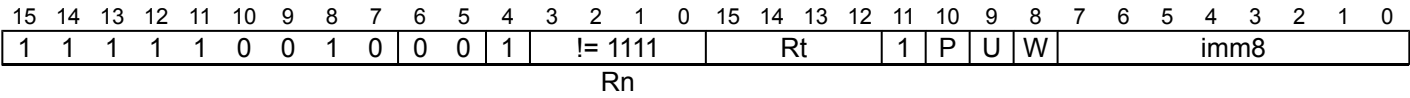
Encoding for the T1 variant

```
LDRSB{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE; constant add = TRUE; constant wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
```

T2



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";
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRSBT";
if P == '0' && W == '0' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// 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 <i>Standard assembler syntax fields</i> .						
<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 <i>LDRSB (literal)</i> .						
+/-	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	+						
+	Specifies the offset is added to the base register.						
<imm>	For encoding A1: 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 encoding T1: 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 encoding T2: 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.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



## 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																															

### Encoding for the A1 variant

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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

## 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											
Rt																															

### Encoding for the T1 variant

```
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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32); constant add = (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 encoding A1: 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 encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant base = Align(PC32, 4);
    constant address = if add then (base + imm32) else (base - imm32);
    R[t] = SignExtend(MemU[address, 1], 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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																															

#### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### 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		

Encoding for the T1 variant

```
LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	1	0	0	0	1	!= 1111			!= 1111			0			0	0	0	0	0	imm2			Rm		
Rn																Rt															

Encoding for the T2 variant

```
LDRSB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.  
For encoding T1 and T2: 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	+
- + Specifies the index register is added to the base register.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    R[t] = SignExtend(MemU[address,1], 32);
    if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>] {, #(<+/-><imm>)}
```

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = FALSE;
constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant m = integer UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

## 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																															

Encoding for the A2 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

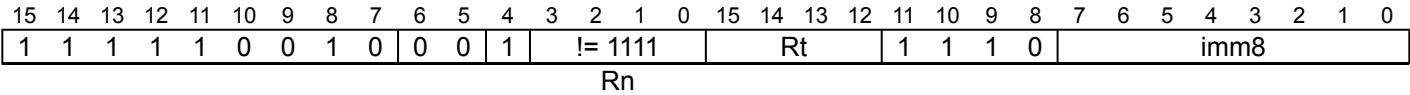
```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE;
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
LDRSBT{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSB (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant postindex = FALSE;  constant add = TRUE;
constant register_form = FALSE;  constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

For encoding A2: 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 offset is added to the base register.
- <imm> For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then R[m] else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if postindex then R[n] else offset_addr;
    R[t] = SignExtend(MemU_unpriv[address,1], 32);
    if postindex then R[n] = offset_addr;

```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	1	imm4L			
cond												Rn																			

#### 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)";
if P == '0' && W == '1' then SEE "LDRSHT";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

## 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													
Rn															Rt																

## Encoding for the T1 variant

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

## Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE; constant add = TRUE; constant wback = 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									
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)";
if Rt == '1111' && P == '1' && U == '0' && W == '0' then SEE "Related instructions";
if P == '1' && U == '1' && W == '0' then SEE "LDRSHT";
if P == '0' && W == '0' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if (t == 15 && W == '1') || (wback && n == t) then UNPREDICTABLE;
// 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":

U	+/-
0	-
1	+
- +

Specifies the offset is added to the base register.
- <imm>

For encoding A1: 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 encoding T1: 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 encoding T2: 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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																																

### Encoding for the A1 variant

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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 || wback then UNPREDICTABLE;
```

## 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 0 0 1								U	0 1		1 1 1 1 1				!= 1111				imm12												
Rt																															

### Encoding for the T1 variant

```
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";
constant t = UInt(Rt); constant imm32 = ZeroExtend(imm12, 32); constant add = (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 encoding A1: 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 encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant base = Align(PC32, 4);
    constant address = if add then (base + imm32) else (base - imm32);
    constant data = MemU[address, 2];
    R[t] = SignExtend(data, 32);
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### 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 <arm-defined-word>unknown</arm-defined-word>. 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		

Encoding for the T1 variant

```
LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	1	0	0	1	1	!= 1111			!= 1111			0			0	0	0	0	0	imm2			Rm		
Rn																Rt															

Encoding for the T2 variant

```
LDRSH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

LDRSH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
if Rt == '1111' then SEE "Related instructions";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant.  
For encoding T1 and T2: is the general-purpose base register, encoded in the "Rn" field.
- +/-  

U	+/-
0	-
1	+

Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U".
- + Specifies the index register is added to the base register.
- <Rm> Is the general-purpose index register, encoded in the "Rm" field.
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    constant data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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																															

#### Encoding for the A1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>] {, # {+/-}<imm>}
```

#### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = FALSE;
constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant m = integer UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

#### 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																															

Encoding for the A2 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

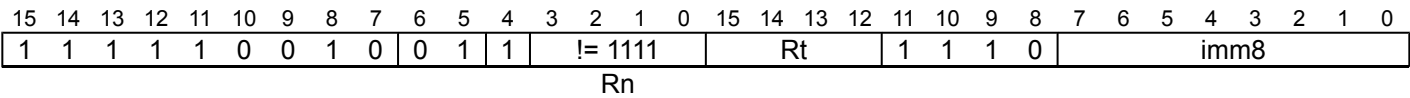
```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE;
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
LDRSHT{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDRSH (literal)";
constant t = UInt(Rt);  constant n = UInt(Rn);  constant postindex = FALSE;  constant add = TRUE;
constant register_form = FALSE;  constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<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 encoding A1: 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	+						

For encoding A2: 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 offset is added to the base register.
- <imm> For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then R[m] else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if postindex then R[n] else offset_addr;
    constant data = MemU_unpriv(address, 2);
    if postindex then R[n] = offset_addr;
    R[t] = SignExtend(data, 32);
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

LDRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1');
constant register_form = FALSE; constant imm32 = ZeroExtend(imm12, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

### 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																															

Encoding for the A2 variant

```
LDRT{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

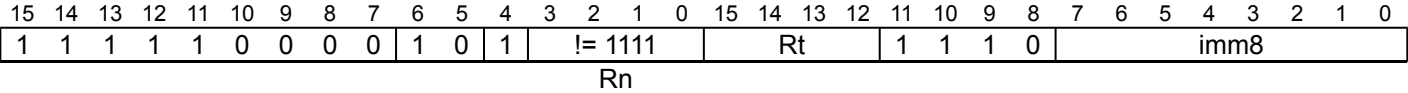
```
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm); constant postindex = TRUE;
constant add = (U == '1'); constant register_form = TRUE;
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
LDRT{<c>}{<q>}<Rt>, [<Rn> {, #<+><imm>}]
```

Decode for this encoding

```
if Rn == '1111' then SEE "LDR (literal)";
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = FALSE; constant add = TRUE;
constant register_form = FALSE; constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.  
For encoding A2 and T1: 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 encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

For encoding A2: 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> 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.
- <imm> For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if postindex then R[n] else offset_addr;
    constant data = MemU_unpriv[address,4];
    if postindex then R[n] = offset_addr;
    R[t] = data;
```

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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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					!= 00000					0	0	0	Rm			
cond				S								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> // (Inside IT block)
```

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			
S																												stype			

#### Encoding for the MOV, shift or rotate by value variant

```
LSL<c>.W {<Rd>}, <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
```

```
LSL{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	<p>For encoding A1: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</p> <p>For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.</p>
<Rm>	<p>For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.</p> <p>For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.</p>
<imm>	<p>For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as &lt;imm&gt; modulo 32.</p> <p>For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as &lt;amount&gt; modulo 32.</p> <p>For encoding T3: is the shift amount, in the range 0 to 31, encoded in the "imm3:imm2" field as &lt;imm&gt; modulo 32.</p>

## Operation

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



## 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				S								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> // (Inside IT block)`

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

`LSL{<c>}{<q>} {<Rd>}, <Rm>, <Rs>`

is equivalent to

`MOV{<c>}{<q>} <Rd>, <Rm>, LSL <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> .
<Rdm>	Is the first general-purpose source register and the destination register, encoded in the "Rdm" field.
<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.

Operation

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

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				S								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>} // (Outside IT block)

is equivalent to

MOVS{<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	0	1	0	1	1	1	1	(0)	imm3			Rd			imm2			0	0	Rm			
S																stype															

Encoding for the MOVS, shift or rotate by value variant

```
LSLS.W {<Rd>, } <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

LSLS{<c>}{<q>} {<Rd>, } <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSL #<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 encoding A1: 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 encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: is the shift amount, in the range 0 to 31, encoded in the "imm5" field as <imm> modulo 32.  For encoding T2: is the shift amount, in the range 1 to 31, encoded in the "imm5" field as <amount> modulo 32.  For encoding T3: 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.

## 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				S								styp																			

#### 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> // (Outside IT block)

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	0	1	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
styp S																																

#### Encoding for the Flag setting variant

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

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

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

Operation

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

## 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				S								styp																			

#### 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> // (Inside IT block)
```

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	(0)	imm3					Rd					imm2					0	1	Rm		
S																styp																			

#### Encoding for the MOV, shift or rotate by value variant

```
LSR<c>.W {<Rd>}, <Rm>, #<imm> // (Inside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)
```

```
LSR{<c>}{<q>} {<Rd>}, <Rm>, #<imm>
```

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a> .  For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.  For encoding T3: 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.



## 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				S								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> // (Inside IT block)`

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

`LSR{<c>}{<q>} {<Rd>}, <Rm>, <Rs>`

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

Operation

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

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											S					styp															

Encoding for the MOV, shift or rotate by value variant

LSRS{<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

LSRS{<q>} {<Rd>}, {<Rm>}, #<imm> // (Outside IT block)

is equivalent to

MOV{<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	0	1	0	1	1	1	1	(0)	imm3			Rd			imm2			0	1	Rm			
S																styp															

Encoding for the MOVS, shift or rotate by value variant

```
LSRS.W {<Rd>, } <Rm>, #<imm> // (Outside IT block, and <Rd>, <Rm>, <imm> can be represented in T2)

LSRS{<c>}{<q>} {<Rd>, } <Rm>, #<imm>
```

is equivalent to

```
MOVS{<c>}{<q>} <Rd>, <Rm>, LSR #<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 encoding A1: 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 encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  For encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1 and T2: is the shift amount, in the range 1 to 32, encoded in the "imm5" field as <imm> modulo 32.  For encoding T3: 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.

## 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				S								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> // (Outside IT block)

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

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

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

Operation

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

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.

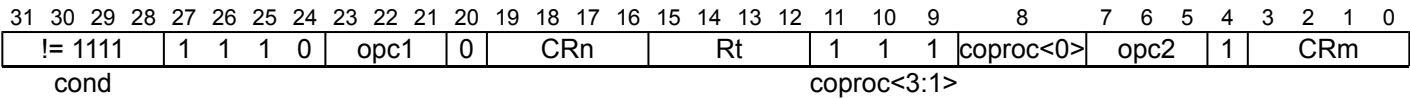
The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

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



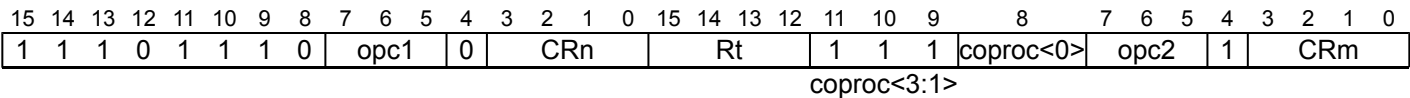
Encoding for the A1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
constant t = UInt(Rt);  constant cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

T1



Encoding for the T1 variant

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

Decode for this encoding

```
constant t = UInt(Rt);  constant cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 “coproc<0>”:

coproc<0>	<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);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



MCCR

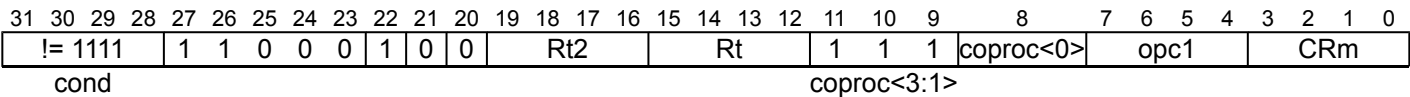
Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register. The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCCR 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 MCCR pseudocode does not show these possible traps.

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

A1



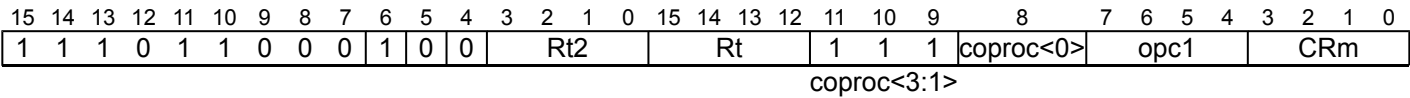
Encoding for the A1 variant

```
MCCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



Encoding for the T1 variant

```
MCCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// 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 “coproc<0>”:

coproc<0>	<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);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
constant setflags = (S == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
Ra																															

### Encoding for the T1 variant

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```
if Ra == '1111' then SEE "MUL";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
constant setflags = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    constant operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    constant addend = SInt(R[a]); // addend = UInt(R[a]) produces the same final results
    constant result = operand1 * operand2 + addend;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged

```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
// 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();
    constant operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    constant operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    constant addend    = SInt(R[a]); // addend    = UInt(R[a]) produces the same final results
    constant result = addend - operand1 * operand2;
    R[d] = result<31:0>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### 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

```
constant d = UInt(Rd); constant setflags = (S == '1');  
constant (imm32, carry) = 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																															

#### Encoding for the A2 variant

```
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

```
constant d = UInt(Rd); constant setflags = FALSE; constant imm32 = ZeroExtend(imm4:imm12, 32);  
constant bit carry = bit UNKNOWN;  
if d == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

```
MOV<c>{<q>} <Rd>, #<imm8> // (Inside IT block)
```

```
MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)
```

### Decode for this encoding

```
constant d = UInt(Rd); constant setflags = !InITBlock(); constant imm32 = ZeroExtend(imm8, 32);  
constant carry = 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	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3			Rd			imm8								

### Encoding for the MOV variant

Applies when (S == 0)

```
MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)
```

```
MOV{<c>}{<q>} <Rd>, #<const>
```

### Encoding for the MOVS variant

Applies when (S == 1)

```
MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)
```

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

### Decode for all variants of this encoding

```
constant d = UInt(Rd); constant setflags = (S == '1');  
constant (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);  
// Armv8-A removes UNPREDICTABLE for R13  
if d == 15 then UNPREDICTABLE;
```

## 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									



## Encoding for the T3 variant

```
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

```
constant d = UInt(Rd); constant setflags = FALSE;
constant imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
constant bit carry = bit UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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 encoding A1: 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 encoding A2, T1, T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<imm8> Is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.

<imm16> For encoding A2: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.  
For encoding T3: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.  
For encoding T2: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant result = imm32;
    if d == 15 then // Can only occur for encoding A1
        if setflags then
            ALUEXCEPTIONRETURN(result);
        else
            ALUWRITEPC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



## 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
!= 1111				0	0	0	1	1	0	1	S	(0)	(0)	(0)	(0)	Rd					imm5					stype	0	Rm			
cond																															

#### 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

```
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = (S == '1');  
constant (shift_t, shift_n) = 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	

## Encoding for the T1 variant

```
MOV{<c>}{<q>} <Rd>, <Rm>
```

## Decode for this encoding

```
constant d = UInt(D:Rd); constant m = UInt(Rm); constant setflags = FALSE;
constant (shift_t, shift_n) = (SRTType_LSL, 0);
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

## 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 for the T2 variant

```
MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block)
```

```
MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)
```

## Decode for this encoding

```
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = !InITBlock();
constant (shift_t, shift_n) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

## 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				

## 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>}.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 r
```

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

## 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.W <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block, and <Rd>, <Rm>, <shift>, <amount> can be re
MOVS{<c>}{<q>} <Rd>, <Rm> {, <shift> #<amount>}
```

Decode for all variants of this encoding

```
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1: 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 encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> For encoding A1 and T1: 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 encoding T2 and T3: is the general-purpose source register, encoded in the "Rm" field.
- <shift> For encoding A1 and T3: 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 encoding T2: 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 encoding A1: 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 encoding T2: 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 encoding T3: 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
<a href="#">ASRS</a> (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; stype == '10'</code>
<a href="#">ASRS</a> (immediate)	T2	<code>op == '10' &amp;&amp; !InITBlock()</code>
<a href="#">ASR</a> (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; stype == '10'</code>
<a href="#">ASR</a> (immediate)	T2	<code>op == '10' &amp;&amp; InITBlock()</code>
<a href="#">LSLS</a> (immediate)	T3 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; imm3:Rd:imm2 != '000xxxx00' &amp;&amp; stype == '00'</code>
<a href="#">LSLS</a> (immediate)	A1 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; imm5 != '00000' &amp;&amp; stype == '00'</code>
<a href="#">LSLS</a> (immediate)	T2	<code>op == '00' &amp;&amp; imm5 != '00000' &amp;&amp; !InITBlock()</code>
<a href="#">LSL</a> (immediate)	T3 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; imm3:Rd:imm2 != '000xxxx00' &amp;&amp; stype == '00'</code>
<a href="#">LSL</a> (immediate)	A1 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; imm5 != '00000' &amp;&amp; stype == '00'</code>
<a href="#">LSL</a> (immediate)	T2	<code>op == '00' &amp;&amp; imm5 != '00000' &amp;&amp; InITBlock()</code>
<a href="#">LSRS</a> (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; stype == '01'</code>
<a href="#">LSRS</a> (immediate)	T2	<code>op == '01' &amp;&amp; !InITBlock()</code>
<a href="#">LSR</a> (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; stype == '01'</code>
<a href="#">LSR</a> (immediate)	T2	<code>op == '01' &amp;&amp; InITBlock()</code>
<a href="#">RORS</a> (immediate)	T3 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; imm3:Rd:imm2 != '000xxxx00' &amp;&amp; stype == '11'</code>
<a href="#">RORS</a> (immediate)	A1 (MOVS, shift or rotate by value)	<code>S == '1' &amp;&amp; imm5 != '00000' &amp;&amp; stype == '11'</code>
<a href="#">ROR</a> (immediate)	T3 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; imm3:Rd:imm2 != '000xxxx00' &amp;&amp; stype == '11'</code>
<a href="#">ROR</a> (immediate)	A1 (MOV, shift or rotate by value)	<code>S == '0' &amp;&amp; imm5 != '00000' &amp;&amp; stype == '11'</code>
<a href="#">RRXS</a>	T3 (MOVS, rotate right with extend)	<code>S == '1' &amp;&amp; imm3 == '000' &amp;&amp; imm2 == '00' &amp;&amp; stype == '11'</code>
<a href="#">RRXS</a>	A1 (MOVS, rotate right with extend)	<code>S == '1' &amp;&amp; imm5 == '00000' &amp;&amp; stype == '11'</code>
<a href="#">RRX</a>	T3 (MOV, rotate right with extend)	<code>S == '0' &amp;&amp; imm3 == '000' &amp;&amp; imm2 == '00' &amp;&amp; stype == '11'</code>
<a href="#">RRX</a>	A1 (MOV, rotate right with extend)	<code>S == '0' &amp;&amp; imm5 == '00000' &amp;&amp; stype == '11'</code>

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = shifted;
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																																	

#### 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

```
constant d = UInt(Rd); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

### 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> // (Inside IT block)
```

```
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> // (Inside IT block)
```

```
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> // (Inside IT block)
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> // (Inside IT block)
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";
constant d = UInt(Rdm); constant m = UInt(Rdm); constant s = UInt(Rs);
constant setflags = !InITBlock(); constant shift_t = 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	stype	S				Rm		1	1	1	1				Rd		0	0	0	0			Rs

## Encoding for the Flag setting variant

Applies when (S == 1)

```
MOVS.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in 16 bits)
MOVS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

## Encoding for the Not flag setting variant

Applies when (S == 0)

```
MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in 16 bits)
MOV{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>
```

## Decode for all variants of this encoding

```
constant d = UInt(Rd); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdm>	Is the general-purpose source register and the destination register, encoded in the "Rdm" field.
<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.

## Alias Conditions

Alias	Of variant	Is preferred when
<a href="#">ASRS (register)</a>	A1 (flag setting)	<code>S == '1' &amp;&amp; stype == '10'</code>
<a href="#">ASRS (register)</a>	T1 (arithmetic shift right)	<code>op == '0100' &amp;&amp; !<a href="#">InITBlock()</a></code>
<a href="#">ASRS (register)</a>	T2 (flag setting)	<code>stype == '10' &amp;&amp; S == '1'</code>
<a href="#">ASR (register)</a>	A1 (not flag setting)	<code>S == '0' &amp;&amp; stype == '10'</code>
<a href="#">ASR (register)</a>	T1 (arithmetic shift right)	<code>op == '0100' &amp;&amp; <a href="#">InITBlock()</a></code>
<a href="#">ASR (register)</a>	T2 (not flag setting)	<code>stype == '10' &amp;&amp; S == '0'</code>
<a href="#">LSLS (register)</a>	A1 (flag setting)	<code>S == '1' &amp;&amp; stype == '00'</code>
<a href="#">LSLS (register)</a>	T1 (logical shift left)	<code>op == '0010' &amp;&amp; !<a href="#">InITBlock()</a></code>
<a href="#">LSLS (register)</a>	T2 (flag setting)	<code>stype == '00' &amp;&amp; S == '1'</code>
<a href="#">LSL (register)</a>	A1 (not flag setting)	<code>S == '0' &amp;&amp; stype == '00'</code>
<a href="#">LSL (register)</a>	T1 (logical shift left)	<code>op == '0010' &amp;&amp; <a href="#">InITBlock()</a></code>
<a href="#">LSL (register)</a>	T2 (not flag setting)	<code>stype == '00' &amp;&amp; S == '0'</code>
<a href="#">LSRS (register)</a>	A1 (flag setting)	<code>S == '1' &amp;&amp; stype == '01'</code>
<a href="#">LSRS (register)</a>	T1 (logical shift right)	<code>op == '0011' &amp;&amp; !<a href="#">InITBlock()</a></code>
<a href="#">LSRS (register)</a>	T2 (flag setting)	<code>stype == '01' &amp;&amp; S == '1'</code>
<a href="#">LSR (register)</a>	A1 (not flag setting)	<code>S == '0' &amp;&amp; stype == '01'</code>
<a href="#">LSR (register)</a>	T1 (logical shift right)	<code>op == '0011' &amp;&amp; <a href="#">InITBlock()</a></code>
<a href="#">LSR (register)</a>	T2 (not flag setting)	<code>stype == '01' &amp;&amp; S == '0'</code>
<a href="#">RORS (register)</a>	A1 (flag setting)	<code>S == '1' &amp;&amp; stype == '11'</code>
<a href="#">RORS (register)</a>	T1 (rotate right)	<code>op == '0111' &amp;&amp; !<a href="#">InITBlock()</a></code>
<a href="#">RORS (register)</a>	T2 (flag setting)	<code>stype == '11' &amp;&amp; S == '1'</code>
<a href="#">ROR (register)</a>	A1 (not flag setting)	<code>S == '0' &amp;&amp; stype == '11'</code>
<a href="#">ROR (register)</a>	T1 (rotate right)	<code>op == '0111' &amp;&amp; <a href="#">InITBlock()</a></code>
<a href="#">ROR (register)</a>	T2 (not flag setting)	<code>stype == '11' &amp;&amp; S == '0'</code>

## Operation

```

if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant (result, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged

```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# MOVT

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

31	30	29	28	27	26	25	24	23	22	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	0	imm4				Rd				imm12											
cond																															

### Encoding for the A1 variant

MOVT{<c>}{<q>} <Rd>, #<imm16>

### Decode for this encoding

```
constant d = UInt(Rd);  constant imm16 = imm4:imm12;
if d == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	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	1	0	0	imm4				0	imm3				Rd				imm8							

### Encoding for the T1 variant

MOVT{<c>}{<q>} <Rd>, #<imm16>

### Decode for this encoding

```
constant d = UInt(Rd);  constant imm16 = imm4:i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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 encoding A1: is a 16-bit unsigned immediate, in the range 0 to 65535, encoded in the "imm4:imm12" field.  
For encoding T1: 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
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MRC

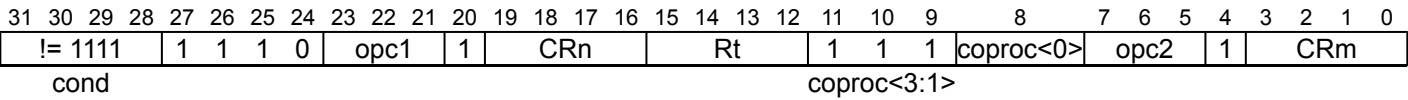
Move to general-purpose register from System register. This instruction copies the value of a System register to a general-purpose register. The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System 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



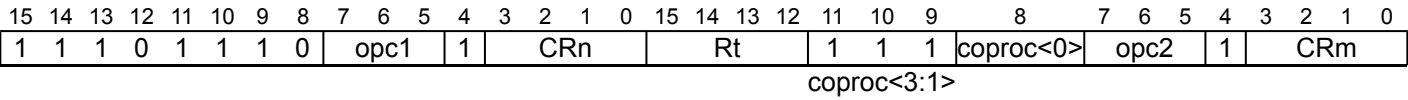
Encoding for the A1 variant

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
constant t = UInt(Rt);  constant cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1



Encoding for the T1 variant

MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}

Decode for this encoding

```
constant t = UInt(Rt);  constant cp = if coproc<0> == '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 “coproc<0>”:

coproc<0>	<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
    UNPREDICTABLE;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	coproc<0>	opc1				CRm			
cond												coproc<3:1>																			

### Encoding for the A1 variant

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

### Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
```

### 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	coproc<0>	opc1				CRm			
																coproc<3:1>															

### Encoding for the T1 variant

MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

### Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE;
```

### 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 “coproc<0>”:

coproc<0>	<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);
```

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

31	30	29	28	27	26	25	24	23	22	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	(1)	(1)	(1)	(1)	Rd				(0)	(0)	0	(0)	0	0	0	0	(0)	(0)	(0)	(0)
cond																															

Encoding for the A1 variant

MRS{<c>}{<q>} <Rd>, <spec\_reg>

Decode for this encoding

```
constant d = UInt(Rd);  constant read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	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	(1)	(1)	(1)	(1)	1	0	(0)	0	Rd				(0)	(0)	0	(0)	(0)	(0)	(0)	(0)

Encoding for the T1 variant

MRS{<c>}{<q>} <Rd>, <spec\_reg>

Decode for this encoding

```
constant d = UInt(Rd);  constant read_spsr = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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
        UNPREDICTABLE;
    else
        R[d] = SPSR\_curr[];
else
    // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
    constant bits(32) mask = '11111000 11101111 00000011 11011111';
    psr_val = GetPSRFromPSTATE(AArch32\_NonDebugState, 32) AND mask;
    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> = bits(1) UNKNOWN;
        psr_val<9:6> = bits(4) UNKNOWN;
        psr_val<4:0> = bits(5) UNKNOWN;
    R[d] = psr_val;
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### Encoding for the A1 variant

MRS{<c>}{<q>} <Rd>, <banked\_reg>

#### Decode for this encoding

```
constant d = UInt(Rd);  constant read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
constant SYSm = 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 for the T1 variant

MRS{<c>}{<q>} <Rd>, <banked\_reg>

#### Decode for this encoding

```
constant d = UInt(Rd);  constant read_spsr = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
constant SYSm = 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”:

<b>R</b>	<b>M</b>	<b>M1</b>	<b>&lt;banked_reg&gt;</b>
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

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        constant mode = PSTATE.M;
        if read_spsr then
            SPSRaccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            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();
                    R[d] = SPSR_mon;
                when '11110' R[d] = SPSR_hyp<31:0>;
            else
                BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
                case SYSm of
                    when '00xxx' // Access the User mode registers
                        constant m = UInt(SYSm<2:0>) + 8;
                        R[d] = Rmode[m,M32\_User];
                    when '01xxx' // Access the FIQ mode registers
                        constant m = UInt(SYSm<2:0>) + 8;
                        R[d] = Rmode[m,M32\_FIQ];
                    when '1000x' // Access the IRQ mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        R[d] = Rmode[m,M32\_IRQ];
                    when '1001x' // Access the Supervisor mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        R[d] = Rmode[m,M32\_Svc];
                    when '1010x' // Access the Abort mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        R[d] = Rmode[m,M32\_Abort];
                    when '1011x' // Access the Undefined mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        R[d] = Rmode[m,M32\_Undef];
                    when '1110x' // Access Monitor registers
                        if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        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];
```

## CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

Encoding for the A1 variant

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

```
constant n = UInt(Rn);  constant write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
constant SYSm = 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 for the T1 variant

```
MSR{<c>}{<q>} <banked_reg>, <Rn>
```

Decode for this encoding

```
constant n = UInt(Rn);  constant write_spsr = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UNPREDICTABLE;
constant SYSm = 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”:

<b>R</b>	<b>M</b>	<b>M1</b>	<b>&lt;banked_reg&gt;</b>
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.



## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        constant mode = PSTATE.M;
        if write_spsr then
            SPSRAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            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();
                    SPSR_mon<31:0> = R[n];
                when '11110' SPSR_hyp<31:0> = R[n];
            else
                BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
                case SYSm of
                    when '00xxx' // Access the User mode registers
                        constant m = UInt(SYSm<2:0>) + 8;
                        Rmode[m,M32\_User] = R[n];
                    when '01xxx' // Access the FIQ mode registers
                        constant m = UInt(SYSm<2:0>) + 8;
                        Rmode[m,M32\_FIQ] = R[n];
                    when '1000x' // Access the IRQ mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        Rmode[m,M32\_IRQ] = R[n];
                    when '1001x' // Access the Supervisor mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        Rmode[m,M32\_Svc] = R[n];
                    when '1010x' // Access the Abort mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        Rmode[m,M32\_Abort] = R[n];
                    when '1011x' // Access the Undefined mode registers
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        Rmode[m,M32\_Undef] = R[n];
                    when '1110x' // Access Monitor registers
                        if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                        constant m = 14 - UInt(SYSm<0>); // LR when SYSm<0> == 0, otherwise SP
                        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];
```

## CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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*.

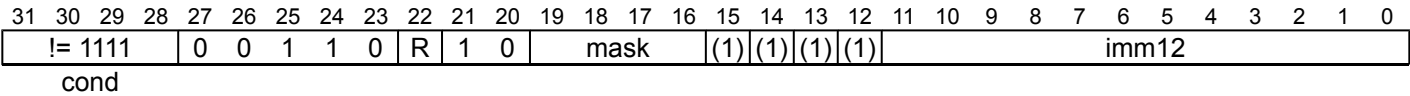
An MSR (immediate) executed in User mode:

- 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



Encoding for the A1 variant

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";
constant imm32 = A32ExpandImm(imm12); constant write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
```

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 nzcvcq, g, or nzcvcg. 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\_nzcvcg 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
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(imm32, mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(imm32, mask);
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

Encoding for the A1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

Decode for this encoding

```
constant n = UInt(Rn);  constant write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE;
```

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 for the T1 variant

```
MSR{<c>}{<q>} <spec_reg>, <Rn>
```

Decode for this encoding

```
constant n = UInt(Rn);  constant write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UNPREDICTABLE;
```

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
            UNPREDICTABLE;
        else
            SPSRWriteByInstr(R[n], mask);
    else
        // Attempts to change to an illegal mode will invoke the Illegal Execution state mechanism
        CPSRWriteByInstr(R[n], mask);
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (Inside IT block)

MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)
```

### Decode for this encoding

```
constant d = UInt(Rdm);  constant n = UInt(Rn);  constant m = UInt(Rdm);
constant setflags = !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			

## Encoding for the T2 variant

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

MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

## Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant setflags = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rdm>	Is the second general-purpose source register holding the multiplier and the destination register, encoded in the "Rdm" field.
<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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    constant operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    constant result = operand1 * operand2;
    R[d] = result<31:0>;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result<31:0>);
        // PSTATE.C, PSTATE.V unchanged
```

## Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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																																

#### 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

```
constant d = UInt(Rd);  constant setflags = (S == '1');  constant a32 = TRUE;
constant bits(12) imm = 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							



## 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

```
constant d = UInt(Rd); constant setflags = (S == '1'); constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1: is the general-purpose destination register, encoded in the "Rd" field.

<const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For encoding T1: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = NOT(imm32);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant setflags = (S == '1');  
constant (shift_t, shift_n) = 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	

Encoding for the T1 variant

```
MVN<c>{<q> <Rd>, <Rm> // (Inside IT block)

MVNS{<q> <Rd>, <Rm> // (Outside IT block)
```

Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 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				

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>.W <Rd>, <Rm> // (Inside IT block, and <Rd>, <Rm> can be represented in T1)

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

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

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

Decode for all variants of this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1 and T2: 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":

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

<amount> For encoding A1: 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 encoding T2: 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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = NOT(shifted);
    if d == 15 then                // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	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				Rs				0	stype	1	Rm				
cond																															

### 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

```
constant d = UInt(Rd); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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
!= 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	0
cond																															

## Encoding for the A1 variant

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

## Encoding for the T1 variant

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

## Encoding for the T2 variant

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
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

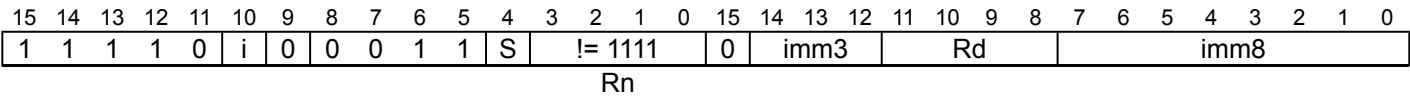
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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



### 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)";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = R[n] OR NOT(imm32);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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	!= 1111			(0)	imm3			Rd			imm2		stype		Rm					
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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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

<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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```

constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1'); constant a32 = TRUE;
constant bits(12) imm = 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							
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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1');
constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the ORRS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm_C(imm, PSTATE.C)
                               else T32ExpandImm_C(imm, PSTATE.C));
    constant result = R[n] OR imm32;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	



Encoding for the T1 variant

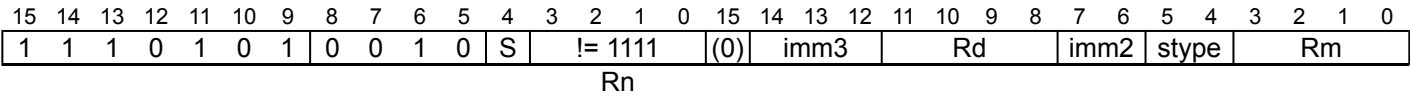
```
ORR<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)

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

Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2



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

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

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

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

Decode for all variants of this encoding

```
if Rn == '1111' then SEE "Related encodings";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<Rdn>	Is the first general-purpose source register and the destination register, encoded in the "Rdn" field.
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"> <li>For the ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li> <li>For the ORRS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li> </ul> For encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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.
----------	---

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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] OR shifted;
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.N = result<31>;
            PSTATE.Z = IsZeroBit(result);
            PSTATE.C = carry;
            // PSTATE.V unchanged

```

Operational information

- If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# 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																															

## 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

```

constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant s = UInt(Rs);
constant setflags = (S == '1');  constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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 general-purpose source register holding a shift amount in its bottom 8 bits, encoded in the "Rs" field.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] OR shifted;
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
        // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant tbform = (tb == '1');
constant (shift_t, shift_n) = DecodeImmShift(tb:'0', imm5);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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			
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;
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant tbform = (tb == '1');
constant (shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: 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 0b000000.

### 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 0b000000. 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 encoding T1: 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 0b000000.

### 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 0b000000. 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();
    constant operand2 = 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>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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	(1)	0	1	1	1	1	1	(1)	(1)	(1)	(1)	imm12											

Encoding for the A1 variant

```
PLD{<c>}{<q>} <label> // (Normal form)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative form)
```

Decode for this encoding

```
constant imm32 = ZeroExtend(imm12, 32); constant add = (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	0	U	0	(0)	1	1	1	1	1	1	1	1	1	1											
																imm12															

Encoding for the T1 variant

```
PLD{<c>}{<q>} <label> // (Preferred syntax)

PLD{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
constant imm32 = ZeroExtend(imm12, 32); constant add = (U == '1');
```

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

Assembler Symbols

- <c>

For encoding A1: see [Standard assembler syntax fields](#). Must be AL or omitted.  
For encoding T1: 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 encoding A1: is the 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.

For encoding T1: 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();
    constant address = if add then (Align(PC32, 4) + imm32) else (Align(PC32, 4) - imm32);
    Hint\_PreloadData(address);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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											
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)";
constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32); constant add = (U == '1');
constant is_pldw = (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											
Rn																															

#### Encoding for the Preload read variant

Applies when (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, #<+><imm>}]
```

#### Encoding for the Preload write variant

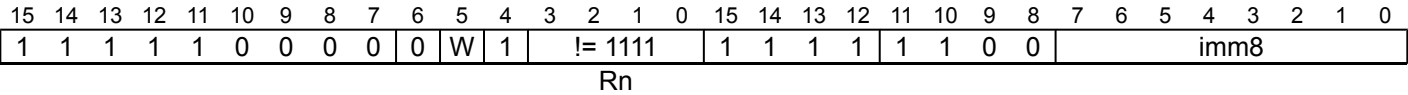
Applies when (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, #<+><imm>}]
```

#### Decode for all variants of this encoding

```
if Rn == '1111' then SEE "PLD (literal)";
constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32); constant add = TRUE;
constant is_pldw = (W == '1');
```

### T2



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)";
constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm8, 32);  constant add = FALSE;
constant is_pldw = (W == '1');
```

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

Assembler Symbols

- <c>For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.  
For encoding T1 and T2: 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)*.
- +/-

U	+/-
0	-
1	+

Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:
- +Specifies the offset is added to the base register.
- <imm>For encoding A1: 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 encoding T1: 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 encoding T2: 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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant address = if add then (R[n] + imm32) else (R[n] - imm32);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

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				

Encoding for the Preload read, optional shift or rotate variant

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

PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]

Encoding for the Preload read, rotate right with extend variant

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

PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]

Encoding for the Preload write, optional shift or rotate variant

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

PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]

Encoding for the Preload write, rotate right with extend variant

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

PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]

Decode for all variants of this encoding

```
constant n = UInt(Rn);  constant m = UInt(Rm);  constant add = (U == '1');
constant is_pldw = (R == '0');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 || (n == 15 && is_pldw) then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	imm2	Rm				

Rn

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)";
constant n = UInt(Rn); constant m = UInt(Rm); constant add = TRUE; constant is_pldw = (W == '1');
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). <c> must be AL or omitted.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> For encoding A1: is the general-purpose base register, encoded in the "Rn" field. The PC can be used.  
For encoding T1: is the general-purpose base register, encoded in the "Rn" field.
- +/-  

U	+/-
0	-
1	+
- + Specifies the index register is added to the base register.
- <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 encoding A1: 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 encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
  constant address = if add then (R[n] + offset) else (R[n] - offset);
  if is_pldw then
    Hint_PreloadDataForWrite(address);
  else
    Hint_PreloadData(address);
```



## 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											

#### Encoding for the A1 variant

```
PLI{<c>}{<q>} [<Rn> {, #<+/-><imm>}]
```

```
PLI{<c>}{<q>} <label> // (Normal form)
```

```
PLI{<c>}{<q>} [PC, #<+/-><imm>] // (Alternative form)
```

#### Decode for this encoding

```
constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32); constant add = (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											
Rn																																

#### Encoding for the T1 variant

```
PLI{<c>}{<q>} [<Rn> {, #<+><imm>}]
```

#### Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";  
constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32); constant add = 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							
Rn																															

#### Encoding for the T2 variant

```
PLI{<c>}{<q>} [<Rn> {, #<-><imm>}]
```

#### Decode for this encoding

```
if Rn == '1111' then SEE "encoding T3";  
constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32); constant add = 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											

Encoding for the T3 variant

```
PLI{<c>}{<q>} <label> // (Preferred syntax)

PLI{<c>}{<q>} [PC, #{+/-}<imm>] // (Alternative syntax)
```

Decode for this encoding

```
constant n = 15;  constant imm32 = ZeroExtend(imm12, 32);  constant add = (U == '1');
```

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

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. Must be AL or omitted.  
For encoding T1, T2 and T3: see *Standard assembler syntax fields*.
  - <q> See *Standard assembler syntax fields*.
  - <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.
  - <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":

U	+/-
0	-
1	+
  - + Specifies the offset is added to the base register.
  - <imm> For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: is a 12-bit unsigned immediate byte offset, in the range 0 to 4095, encoded in the "imm12" field.
- 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();
    constant base = if n == 15 then Align(PC32, 4) else R[n];
    constant address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

# 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				

### 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

```
constant n = UInt(Rn);  constant m = UInt(Rm);  constant add = (U == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	imm2	Rm				
Rn																															

### Encoding for the T1 variant

PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

### Decode for this encoding

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
constant n = UInt(Rn);  constant m = UInt(Rm);  constant add = TRUE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). <c> must be AL or omitted.  
For encoding T1: 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 index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in “U”:

U	+/-
0	-
1	+
- +

Specifies the index register is added to the base register.
- <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 encoding A1: 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 encoding T1: is the shift amount, in the range 0 to 3, defaulting to 0 and encoded in the "imm2" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant address = if add then (R[n] + offset) else (R[n] - offset);
    Hint\_PreloadInstr(address);
```

# 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							

### Encoding for the T1 variant

```
POP{<c>}{<q>} <registers> // (Preferred syntax)

LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

### Decode for this encoding

```
constant registers = P:'0000000':register_list;    constant UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
if registers<15> == '1' && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

### 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();
    address = R[13];
    for i = 0 to 14
        if registers<i> == '1' then
            R[i] = if UnalignedAllowed then MemU[address,4] else MemA[address,4];
            address = address + 4;
    if registers<15> == '1' then
        if UnalignedAllowed then
            if address<1:0> == '00' then
                LoadWritePC(MemU[address,4]);
            else
                UNPREDICTABLE;
        else
            LoadWritePC(MemA[address,4]);
    if registers<13> == '0' then R[13] = R[13] + 4*BitCount(registers);
    if registers<13> == '1' then R[13] = bits(32) UNKNOWN;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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				W				Rn																							

### Encoding for the A1 variant

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<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													
W																Rn															

### Encoding for the T2 variant

```
POP{<c>}.W <registers> // (All registers in R0-R7, PC)
```

```
POP{<c>}{<q>} <registers>
```

is equivalent to

```
LDM{<c>}{<q>} 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 encoding A1: 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 encoding T2: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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
!= 1111				0	1	0	0	1	0	0	1	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0
cond				P		U		W		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	0	0	0	0	1	0	1	1	1	0	1	Rt				1	0	1	1	0	0	0	0	0	1	0	0
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 }.
- <Rt> For encoding A1: 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 encoding T4: 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](#).

## Operation

The description of [LDR \(immediate\)](#) gives the operational pseudocode 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

Encoding for the A1 variant

```
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

Encoding for the T1 variant

```
PSSBB{<q>}
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

Assembler Symbols

<q> See *Standard assembler syntax fields*.

Operation

```
if ConditionPassed() then
    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							

### Encoding for the T1 variant

```
PUSH{<c>}{<q>} <registers> // (Preferred syntax)

STMDB{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

### Decode for this encoding

```
constant registers = '0':M:'000000':register_list; constant UnalignedAllowed = FALSE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

### 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();
    address = R[13] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == 13 && i != LowestSetBit(registers) then // Only possible for encoding A1
                MemA[address,4] = bits(32) UNKNOWN;
            else
                if UnalignedAllowed then
                    MemU[address,4] = R[i];
                else
                    MemA[address,4] = R[i];
                address = address + 4;
    if registers<15> == '1' then // Only possible for encoding A1 or A2
        if UnalignedAllowed then
            MemU[address,4] = PCStoreValue();
        else
            MemA[address,4] = PCStoreValue();
    R[13] = R[13] - 4*BitCount(registers);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

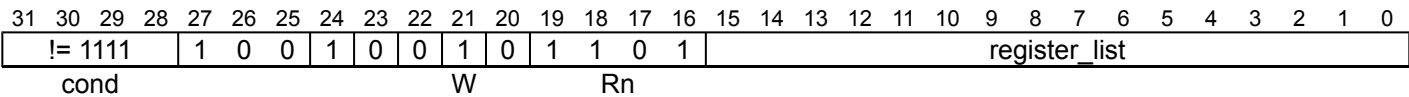
# PUSH (multiple registers)

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



### Encoding for the A1 variant

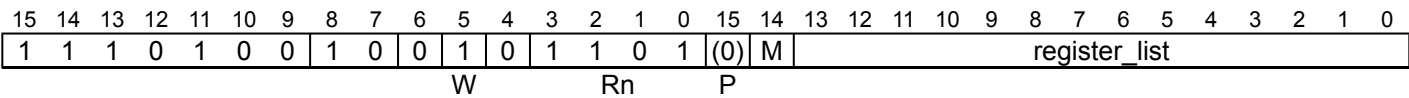
```
PUSH{<c>}{<q>} <registers>
```

is equivalent to

```
STMDB{<c>}{<q>} SP!, <registers>
```

and is the preferred disassembly when `BitCount(register_list) > 1`.

## T1



### Encoding for the T1 variant

```
PUSH{<c>}.W <registers> // (All registers in R0-R7, LR)
```

```
PUSH{<c>}{<q>} <registers>
```

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 encoding A1: 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 encoding T1: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	0	1	0	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0
cond				P		U	W		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	Rt				1	1	0	1	0	0	0	0	0	1	0	0
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 }.
- <Rt>

For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.  
For encoding T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

Operation

The description of [STR \(immediate\)](#) gives the operational pseudocode 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																															

Encoding for the A1 variant

QADD{<c>}{<q>} {<Rd>,<Rm>,<Rn>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

QADD{<c>}{<q>} {<Rd>,<Rm>,<Rn>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    boolean sat;
    (R[d], sat) = SignedSatQ(SInt(R[m]) + SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```



# 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																															

### Encoding for the A1 variant

QADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

QADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    constant sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0> = SignedSat(sum1, 16);
    R[d]<31:16> = SignedSat(sum2, 16);
```



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																															

Encoding for the A1 variant

QADD8{<c>}{<q>} {<Rd>,<Rn>,<Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

QADD8{<c>}{<q>} {<Rd>,<Rn>,<Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    constant sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    constant sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    constant sum4 = SInt(R[n]<31:24>) + SInt(R[m]<31:24>);
    R[d]<7:0> = SignedSat(sum1, 8);
    R[d]<15:8> = SignedSat(sum2, 8);
    R[d]<23:16> = SignedSat(sum3, 8);
    R[d]<31:24> = SignedSat(sum4, 8);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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	1	1	Rm			
cond																															

### Encoding for the A1 variant

QASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	Rm			

### Encoding for the T1 variant

QASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    constant sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>    = SignedSat(diff, 16);
    R[d]<31:16>   = SignedSat(sum, 16);
```





# 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																															

### Encoding for the A1 variant

QDADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

QDADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    boolean sat2;
    (R[d], sat2) = SignedSatQ(SInt(R[m]) + SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```



# 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																															

### Encoding for the A1 variant

QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

QDSUB{<c>}{<q>} {<Rd>,} <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant (doubled, sat1) = SignedSatQ(2 * SInt(R[n]), 32);
    boolean sat2;
    (R[d], sat2) = SignedSatQ(SInt(R[m]) - SInt(doubled), 32);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```



# 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

31	30	29	28	27	26	25	24	23	22	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	0	1	Rm			
cond																															

### Encoding for the A1 variant

QSAX{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	Rm			

### Encoding for the T1 variant

QSAX{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    constant diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>    = SignedSat(sum, 16);
    R[d]<31:16>   = SignedSat(diff, 16);
```



# 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																															

### Encoding for the A1 variant

QSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

QSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    boolean sat;
    (R[d], sat) = SignedSatQ(SInt(R[m]) - SInt(R[n]), 32);
    if sat then
        PSTATE.Q = '1';
```





# 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																															

### Encoding for the A1 variant

QSUB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

QSUB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    constant diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0> = SignedSat(diff1, 16);
    R[d]<31:16> = SignedSat(diff2, 16);
```



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																															

Encoding for the A1 variant

QSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

QSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    constant diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    constant diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    constant diff4 = SInt(R[n]<31:24>) - SInt(R[m]<31:24>);
    R[d]<7:0> = SignedSat(diff1, 8);
    R[d]<15:8> = SignedSat(diff2, 8);
    R[d]<23:16> = SignedSat(diff3, 8);
    R[d]<31:24> = SignedSat(diff4, 8);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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)	(1)	Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rm			
cond																															

### Encoding for the A1 variant

RBIT{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

RBIT{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UNPREDICTABLE;
```

## 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 encoding A1: is the general-purpose source register, encoded in the "Rm" field.  
For encoding T1: 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();
    bits(32) result;
    for i = 0 to 31
        result<31-i> = R[m]<i>;
    R[d] = result;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

REV{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

REV{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = 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			

### Encoding for the T2 variant

REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REV{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.  For encoding T2: 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();
    bits(32) result;
    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;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

REV16{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

REV16{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = 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			

### Encoding for the T2 variant

REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REV16{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field.  For encoding T2: 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();
    bits(32) result;
    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;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)	(1)	Rd				(1)	(1)	(1)	(1)	1	0	1	1	Rm			
cond																															

### Encoding for the A1 variant

REVSH{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

REVSH{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = 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			

### Encoding for the T2 variant

REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REVSH{<c>}{<q>} <Rd>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if m != n || d == 15 || m == 15 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	Is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1 and T1: is the general-purpose source register, encoded in the "Rm" field. For encoding T2: 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();
    bits(32) result;
    result<31:8> = SignExtend(R[m]<7:0>, 24);
    result<7:0> = R[m]<15:8>;
    R[d] = result;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)

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

```
constant n = UInt(Rn);
constant wback = (W == '1');  constant increment = (U == '1');  constant wordhigher = (P == U);
if n == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	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)

Encoding for the T1 variant

```
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

```
constant n = UInt(Rn); constant wback = (W == '1'); constant increment = FALSE;
constant wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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				Rn	(1)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

Encoding for the T2 variant

```
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

```
constant n = UInt(Rn); constant wback = (W == '1'); constant increment = TRUE;
constant wordhigher = FALSE;
if n == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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

Assembler Symbols

- IA

For encoding A1: is an optional suffix to indicate the Increment After variant.

For encoding T2: is an optional suffix for the Increment After form.
- <c>

For encoding A1: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

For encoding T1 and T2: 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.
- 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
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        address = if increment then R[n] else R[n]-8;
        if wordhigher then address = address+4;
        constant new_pc_value = MemA[address,4];
        constant spsr = MemA[address+4,4];
        if wback then R[n] = if increment then R[n]+8 else R[n]-8;
        AArch32.ExceptionReturn(new_pc_value, spsr);
```

## CONSTRAINED UNPREDICTABLE behavior

If `PSTATE.EL == EL0`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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					!= 00000					1	1	0	Rm			
cond				S								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			
S																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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a> . For encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
<Rm>	For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T3: is the general-purpose source register, encoded in the "Rm" field.
<imm>	For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field. For encoding T3: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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				S								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>}` // (Inside IT block)

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

`ROR{<c>}{<q>}{<Rd>}, {<Rm>, <Rs>}`

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

Operation

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

## 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				S								imm5					stype															

#### Encoding for the MOV, shift or rotate by value variant

RORS{<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	1	1	1	1	1	(0)	imm3			Rd			imm2			1	1	Rm			
S																stype															

#### Encoding for the MOV, shift or rotate by value variant

Applies when (!(imm3 == 000 && imm2 == 00))

RORS{<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 encoding A1: 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 encoding T3: is the general-purpose destination register, encoded in the "Rd" field.

<Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: is the general-purpose source register, encoded in the "Rm" field.

<imm> For encoding A1: is the shift amount, in the range 1 to 31, encoded in the "imm5" field.

For encoding T3: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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				S												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> // (Outside IT block)

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

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

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

Operation

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

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				S								imm5					stype														

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			
S																imm3				imm2				stype							

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 encoding A1: 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 encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
- <Rm>For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T3: 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.





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				S								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	(0)	0	0	0	Rd				0		0	1		1	Rm		
S												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 encoding A1: 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 encoding T3: is the general-purpose destination register, encoded in the "Rd" field.
- <Rm> For encoding A1: is the general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T3: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant imm32 = 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	

### Encoding for the T1 variant

RSB<[<c>](#)>{<[q](#)>} {<Rd>,> <Rn>,> #0 // (Inside IT block)

RSBS<[<q>](#)> {<Rd>,> <Rn>,> #0 // (Outside IT block)

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = !InITBlock();
constant imm32 = Zeros(32); // 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							

## Encoding for the RSB variant

Applies when (S == 0)

```
RSB<c>.W {<Rd>}, <Rn>, #0 // (Inside IT block)

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

## Encoding for the RSBS variant

Applies when (S == 1)

```
RSBS.W {<Rd>}, <Rn>, #0 // (Outside IT block)

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

## Decode for all variants of this encoding

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the RSB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the RSBS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1 and T2: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T2: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), imm32, '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUEXCEPTIONRETURN(result);
        else
            ALUWRITEPC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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			

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

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

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 encoding A1: 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>.For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated. For encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), shifted, '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant s = UInt(Rs);
constant setflags = (S == '1');  constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), shifted, '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant imm32 = 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 *PSTATE* 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();
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), imm32, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    if d == 15 then
        if setflags then
            ALUEXCEPTIONReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;

```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant s = UInt(Rs);
constant setflags = (S == '1');  constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(NOT(R[n]), shifted, PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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	0	1	Rm			
cond																															

### Encoding for the A1 variant

SADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	0	Rm			

### Encoding for the T1 variant

SADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    constant sum2 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    constant sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    constant sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    constant sum4 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    constant sum  = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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)

Encoding for the A1 variant

```
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)

Encoding for the T1 variant

```
SB{<q>}
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

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();
```



# 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																															

## 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1');
constant imm32 = 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							

## 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

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

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A1: 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: <ul style="list-style-type: none"><li>For the SBC variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li><li>For the SBCS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li></ul> For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(imm32), PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



## 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																															

#### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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	

Encoding for the T1 variant

```
SBC<c>{<q>} {<Rdn>}, <Rdn>, <Rm> // (Inside IT block)

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

Decode for this encoding

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 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			

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

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

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

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

Decode for all variants of this encoding

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

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

<Rd> For encoding A1: 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 encoding T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.

<Rn> For encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.

For encoding T2: is the first general-purpose source register, encoded in the "Rn" field.

<Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.

For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

### Operational information

- If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant setflags = (S == '1'); constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), PSTATE.C);
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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																															

### Encoding for the A1 variant

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

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);
constant integer lsbit = UInt(lsb);
constant integer widthminus1 = UInt(widthm1);
constant integer msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

### 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			

### Encoding for the T1 variant

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

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);
constant integer lsbit = UInt(imm3:imm2);
constant integer widthminus1 = UInt(widthm1);
constant integer msbit = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: 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(R[n]<msbit:lsbit>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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												Ra																			

### Encoding for the A1 variant

SDIV{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

### 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			
Ra																															

### Encoding for the T1 variant

SDIV{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 <Rd> 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();
    constant integer dividend = SInt(R[n]);
    constant integer divisor  = SInt(R[m]);
    integer result;
    if divisor == 0 then
        result = 0;
    elsif (dividend < 0) == (divisor < 0) then
        result = Abs(dividend) DIV Abs(divisor);    // same signs - positive result
    else
        result = -(Abs(dividend) DIV Abs(divisor)); // different signs - negative result
    R[d] = result<31:0>;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 for the A1 variant

SEL{<c>}{<q>} {<Rd>,,} <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

SEL{<c>}{<q>} {<Rd>,,} <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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>;
```



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)

Encoding for the A1 variant

```
SETEND{<q>} <endian_specifier> // (Cannot be conditional)
```

Decode for this encoding

```
constant set_bigend = (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)

Encoding for the T1 variant

```
SETEND{<q>} <endian_specifier> // (Not permitted in IT block)
```

Decode for this encoding

```
constant set_bigend = (E == '1');  
if InITBlock() then UNPREDICTABLE;
```

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 for the A1 variant

```
SETPAN{<q>} #<imm> // (Cannot be conditional)
```

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_PAN) then UNDEFINED;
constant value = 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 for the T1 variant

```
SETPAN{<q>} #<imm> // (Not permitted in IT block)
```

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_PAN) then UNDEFINED;
constant value = 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;
```



# 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	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0
cond																															

### Encoding for the A1 variant

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

### Encoding for the T1 variant

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

### Encoding for the T2 variant

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();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

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

Encoding for the T1 variant

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

Encoding for the T2 variant

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();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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	0	1	Rm			
cond																															

### Encoding for the A1 variant

SHADD16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

SHADD16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    constant sum2 = SInt(R[n]<31:16>) + SInt(R[m]<31:16>);
    R[d]<15:0> = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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	0	0	1	Rm			
cond																															

### Encoding for the A1 variant

SHADD8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

SHADD8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    constant sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    constant sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    constant sum4 = 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>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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																															

### Encoding for the A1 variant

SHASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SHASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    constant sum  = SInt(R[n]<31:16>) + SInt(R[m]<15:0>);
    R[d]<15:0>    = diff<16:1>;
    R[d]<31:16>   = sum<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SHSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SHSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    constant diff = SInt(R[n]<31:16>) - SInt(R[m]<15:0>);
    R[d]<15:0>    = sum<16:1>;
    R[d]<31:16>   = diff<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SHSUB16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SHSUB16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    constant diff2 = SInt(R[n]<31:16>) - SInt(R[m]<31:16>);
    R[d]<15:0> = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SHSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SHSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    constant diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    constant diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    constant diff4 = 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>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



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)	0	1	1	1	imm4			
cond																															

Encoding for the A1 variant

```
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)

Encoding for the T1 variant

```
SMC{<c>}{<q>} {#}<imm4>
```

Decode for this encoding

```
// imm4 is for assembly/disassembly only and is ignored by hardware
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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;
        else
            if SCR.SCD == '1' then
                // SMC disabled
                if CurrentSecurityState() == SS\_Secure then
                    // Executes either as a NOP or UNALLOCATED.
                    constant c = ConstrainUnpredictable(Unpredictable\_SMD);
                    assert c IN {Constraint\_NOP, Constraint\_UNDEF};
                    if c == Constraint\_NOP then ExecuteAsNOP();
                    UNDEFINED;
            else
                if !ELUsingAArch32(EL3) then
                    AArch64.CallSecureMonitor(Zeros(16));
                else
                    AArch32.TakeSMCException();
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
constant n_high = (N == '1');  constant m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant a = UInt(Ra);
constant n_high = (N == '1'); constant m_high = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
constant operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
constant operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
constant result = SInt(operand1) * SInt(operand2) + SInt(R[a]);
R[d] = result<31:0>;
if result != SInt(result<31:0>) then // Signed overflow
    PSTATE.Q = '1';
```

# 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												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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant a = UInt(Ra);
constant m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm); constant a = UInt(Ra);
constant m_swap = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = product1 + product2 + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then // Signed overflow
        PSTATE.Q = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant result = 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(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



# 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																															

### 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);
constant n_high = (N == '1');  constant m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

## 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

```
constant dLo = UInt(RdLo); constant dHi = UInt(RdHi); constant n = UInt(Rn);
constant m = UInt(Rm);
constant n_high = (N == '1'); constant m_high = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: 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 encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    constant operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    constant result = SInt(operand1) * SInt(operand2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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  $2^{64}$ . 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	0	M	1	Rn			
cond																															

## 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

## 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

```
constant dLo = UInt(RdLo); constant dHi = UInt(RdHi); constant n = UInt(Rn);
constant m = UInt(Rm); constant m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = product1 + product2 + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



# 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

31	30	29	28	27	26	25	24	23	22	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				Ra				Rm				1	M	0	0	Rn			
cond																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant a = UInt(Ra);  constant m_high = (M == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				!= 1111				Rd				0	0	0	M	Rm			
Ra																															

### 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";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant a = UInt(Ra);  constant m_high = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    constant result = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	Rd				!= 1111				Rm				0	1	M	1	Rn			
cond												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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant a = UInt(Ra); constant m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant a = UInt(Ra); constant m_swap = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = (product1 - product2) + SInt(R[a]);
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then // Signed overflow
        PSTATE.Q = '1';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SMLSLD, SMLSLDX

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																															

### Encoding for the SMLSLD variant

Applies when (M == 0)

SMLSLD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Encoding for the SMLSLDX variant

Applies when (M == 1)

SMLSLDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for all variants of this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

## Encoding for the SMLS LD variant

Applies when (M == 0)

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

## Encoding for the SMLS LD X variant

Applies when (M == 1)

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

## Decode for all variants of this encoding

```
constant dLo = UInt(RdLo); constant dHi = UInt(RdHi); constant n = UInt(Rn);
constant m = UInt(Rm); constant m_swap = (M == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = (product1 - product2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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												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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant a = UInt(Ra); constant round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
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";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant a = UInt(Ra); constant round = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    integer result = (SInt(R[a]) << 32) + SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant a = UInt(Ra);  constant round = (R == '1');
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant a = UInt(Ra);  constant round = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    integer result = (SInt(R[a]) << 32) - SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant round = (R == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant round = (R == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    integer result = SInt(R[n]) * SInt(R[m]);
    if round then result = result + 0x80000000;
    R[d] = result<63:32>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_swap = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = product1 + product2;
    R[d] = result<31:0>;
    if result != SInt(result<31:0>) then // Signed overflow
        PSTATE.Q = '1';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant n_high = (N == '1');  constant m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

## 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant n_high = (N == '1'); constant m_high = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    constant operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    constant result = SInt(operand1) * SInt(operand2);
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```



## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant result = 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(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_high = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_high = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    constant product = SInt(R[n]) * SInt(operand2);
    R[d] = product<47:16>;
    // Signed overflow cannot occur
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_swap = (M == '1');
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant m_swap = (M == '1');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant operand2 = if m_swap then ROR(R[m],16) else R[m];
    constant product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    constant product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    constant result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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				

### 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

```
constant wback = (W == '1');  constant increment = (U == '1');  constant wordhigher = (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						

### Encoding for the T1 variant

SRSDB{<c>}{<q>} SP{!}, #<mode>

### Decode for this encoding

```
constant wback = (W == '1');  constant increment = FALSE;  constant wordhigher = 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

Encoding for the T2 variant

```
SRS{IA}{<c>}{<q>} SP{!}, #<mode>
```

Decode for this encoding

```
constant wback = (W == '1');  constant increment = TRUE;  constant wordhigher = 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

- IA

For encoding A1: is an optional suffix to indicate the Increment After variant.

For encoding T2: is an optional suffix for the Increment After form.
- <c>

For encoding A1: see [Standard assembler syntax fields](#). <c> must be AL or omitted.

For encoding T1 and T2: 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](#).
- 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;

    // 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
      UNPREDICTABLE;
    elsif mode == M32\_Hyp then          // Check for attempt to access Hyp mode SP
      UNPREDICTABLE;
    elsif mode == M32\_Monitor then      // Check for attempt to access Monitor mode SP
      if !HaveEL(EL3) || CurrentSecurityState() != SS\_Secure then
        UNPREDICTABLE;
      elsif !ELUsingAArch32(EL3) then
        AArch64.MonitorModeTrap();
    elsif BadMode(mode) then
      UNPREDICTABLE;

    constant base = Rmode[13,mode];
    address = if increment then base else base-8;
    if wordhigher then address = address+4;
    MemA[address,4] = LR;
    MemA[address+4,4] = SPSR\_curr[];
    if wback then Rmode[13,mode] = if increment then base+8 else base-8;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      if PSTATE.EL == EL2 then          // UNDEFINED at EL2
        UNDEFINED;

      // 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
        UNPREDICTABLE;
      elsif mode == M32\_Hyp then          // Check for attempt to access Hyp mode SP
        UNPREDICTABLE;
      elsif mode == M32\_Monitor then      // Check for attempt to access Monitor mode SP
        if !HaveEL(EL3) || CurrentSecurityState() != SS\_Secure then
          UNPREDICTABLE;
        elsif !ELUsingAArch32(EL3) then
          AArch64.MonitorModeTrap();
      elsif BadMode(mode) then
        UNPREDICTABLE;

      constant base = Rmode[13,mode];
      address = if increment then base else base-8;
      if wordhigher then address = address+4;
      MemA[address,4] = LR;
      MemA[address+4,4] = SPSR\_curr[];
      if wback then Rmode[13,mode] = if increment then base+8 else base-8;
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant d = UInt(Rd); constant n = UInt(Rn); constant saturate_to = UInt(sat_imm)+1;
constant (shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	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) == '00000' then SEE "SSAT16";
constant d = UInt(Rd); constant n = UInt(Rn); constant saturate_to = UInt(sat_imm)+1;
constant (shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
```

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.
<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 encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32. For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant operand = Shift(R[n], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    constant (result, sat) = SignedSatQ(SInt(operand), saturate_to);
    R[d] = SignExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

# 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																															

### Encoding for the A1 variant

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	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			

### Encoding for the T1 variant

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm)+1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    constant (result2, sat2) = SignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = SignExtend(result1, 16);
    R[d]<31:16> = SignExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```



# 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																															

### Encoding for the A1 variant

SSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    constant diff = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

### Encoding for the A1 variant

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

### Encoding for the T1 variant

SSBB{<q>}

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
```

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();
    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																															

### Encoding for the A1 variant

SSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

SSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    constant diff2 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

SSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

SSUB8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    constant diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    constant diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    constant diff4 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### 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;
constant n = UInt(Rn); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32); constant index = (P == '1');
constant add = (U == '1'); constant wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

## 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							

### 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;
constant n = UInt(Rn); constant cp = 14;
constant imm32 = ZeroExtend(imm8:'00', 32); constant index = (P == '1');
constant add = (U == '1'); constant wback = (W == '1');
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rn>	For the offset or unindexed variant: is the general-purpose base register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For the offset, post-indexed or pre-indexed variant: is the general-purpose base register, encoded in the "Rn" field.
<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.
+/-	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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = 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;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

STL{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

STL{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    MemO[address, 4] = R[t];
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





# 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																															

### Encoding for the A1 variant

STLB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

STLB{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    MemO[address, 1] = R[t]<7:0>;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# 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																															

### Encoding for the A1 variant

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### 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			

### Encoding for the T1 variant

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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();
    constant address = R[n];
    if AArch32.ExclusiveMonitorsPass(address, 4) then
        MemO[address, 4] = R[t];
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### 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			

### Encoding for the T1 variant

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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();
    constant address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemO[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

Encoding for the A1 variant

STLEXD{<c>}{<q>}<Rd>, <Rt>, <Rt2>, [<Rn>]

Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = t + 1;
constant d = UInt(Rd);  constant n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

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			



## Encoding for the T1 variant

```
STLEXD{<c>}{<q> <Rd>, <Rt>, <Rt2>, [<Rn>]}
```

## Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant n = UInt(Rn);  
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;  
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> If the operation fails to update memory.
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. <Rt> must be even-numbered and not R14. For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. <Rt2> must be <R(t+1)>. For encoding T1: 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();
    constant address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    constant value = if BigEndian(AccessType GPR) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        MemO[address, 8] = value;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

STLEXH{<c>}{<q>}<Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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	1	Rd			

Encoding for the T1 variant

STLEXH{<c>}{<q>}<Rd>, <Rt>, [<Rn>]

Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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();
    constant address = R[n];
    if AArch32.ExclusiveMonitorsPass(address, 2) then
        MemO[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

STLH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

STLH{<c>}{<q>} <Rt>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn);
if t == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant address = R[n];
    MemO[address, 2] = R[t]<15:0>;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



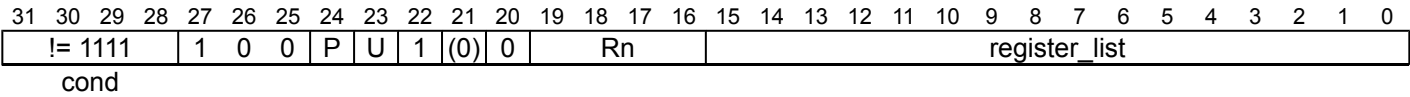
# STM (User registers)

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 for the A1 variant

```
STM{<amode>}{<c>}{<q>} <Rn>, <registers>^
```

## Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list;
constant increment = (U == '1'); constant wordhigher = (P == U);
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

## 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;
    elseif PSTATE.M IN {M32_User, M32_System} then
        UNPREDICTABLE;
    else
        constant length = 4*BitCount(registers);
        address = if increment then R[n] else R[n]-length;
        if wordhigher then address = address+4;
        for i = 0 to 14
            if registers<i> == '1' then // Store User mode register
                MemS[address,4] = Rmode[i, M32_User];
                address = address + 4;
        if registers<15> == '1' then
            MemS[address,4] = PCStoreValue();
```

## 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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

```
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

## 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						

### Encoding for the T1 variant

```
STM{IA}{<c>}{<q>} <Rn>!, <registers> // (Preferred syntax)

STMEA{<c>}{<q>} <Rn>!, <registers> // (Alternate syntax, Empty Ascending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = '00000000':register_list; constant wback = TRUE;
if BitCount(registers) < 1 then UNPREDICTABLE;
```

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	register_list														
P																															

Encoding for the T2 variant

```
STM{IA}{<c>}.W <Rn>{!}, <registers> // (Preferred syntax, if <Rn>, '!' and <registers> can be represented
STMEA{<c>}.W <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack, if <Rn>, '!' and <registers> can be represented
STM{IA}{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)
STMEA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Empty Ascending stack)
```

Decode for this encoding

```
constant n = UInt(Rn); constant registers = P:M:register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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>	<p>For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.</p> <p>The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.</p> <p>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.</p> <p>For encoding T1: 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.</p> <p>For encoding T2: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.</p> <p>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.</p>

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN; // Only possible for encodings T1 and A1
            else
                MemS[address,4] = R[i];
            address = address + 4;
    if registers<15> == '1' then // Only possible for encoding A1
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

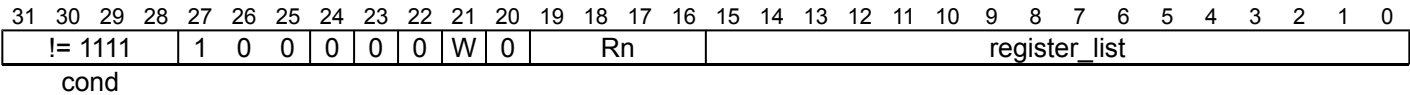
STMDA, STMED

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 for the A1 variant

```
STMDA{<c>}{<q>}<Rn>{!}, <registers> // (Preferred syntax)

STMED{<c>}{<q>}<Rn>{!}, <registers> // (Alternate syntax, Empty Descending stack)
```

Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

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 <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>	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();
    address = R[n] - 4*BitCount(registers) + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
                address = address + 4;
    if registers<15> == '1' then
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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.

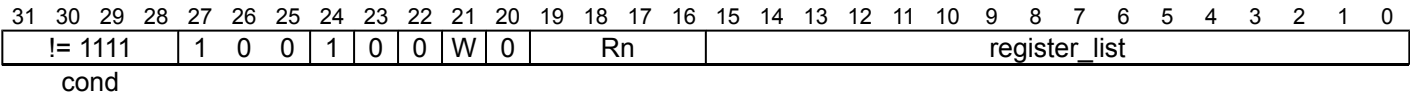
The lowest-numbered register is loaded from the lowest memory address, through to the highest-numbered register from the highest memory address. See also *Encoding of lists of general-purpose registers and the PC*.

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



### Encoding for the A1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

### Decode for this encoding

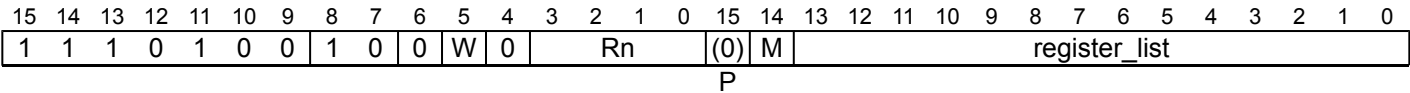
```
constant n = UInt(Rn);  constant registers = register_list;  constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

## 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



### Encoding for the T1 variant

```
STMDB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFD{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Descending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn);  constant registers = P:M:register_list;  constant wback = (W == '1');
if n == 15 || BitCount(registers) < 2 then UNPREDICTABLE;
if wback && registers<n> == '1' then UNPREDICTABLE;
if registers<13> == '1' then UNPREDICTABLE;
if registers<15> == '1' then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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>	<p>For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.</p> <p>The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.</p> <p>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.</p> <p>For encoding T1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }.</p> <p>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.</p>

Alias Conditions

Alias	Of variant Is preferred when	
<a href="#">PUSH (multiple registers)</a>	T1	<code>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(M:register_list) &gt; 1</code>
<a href="#">PUSH (multiple registers)</a>	A1	<code>W == '1' &amp;&amp; Rn == '1101' &amp;&amp; BitCount(register_list) &gt; 1</code>

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n] - 4*BitCount(registers);
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN; // Only possible for encoding A1
            else
                MemS[address,4] = R[i];
                address = address + 4;
    if registers<15> == '1' then // Only possible for encoding A1
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] - 4*BitCount(registers);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



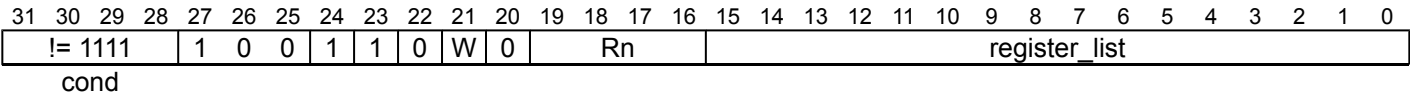
# STMIB, STMFA

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 for the A1 variant

```
STMIB{<c>}{<q>} <Rn>{!}, <registers> // (Preferred syntax)

STMFA{<c>}{<q>} <Rn>{!}, <registers> // (Alternate syntax, Full Ascending stack)
```

### Decode for this encoding

```
constant n = UInt(Rn); constant registers = register_list; constant wback = (W == '1');
if n == 15 || BitCount(registers) < 1 then UNPREDICTABLE;
```

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    address = R[n] + 4;
    for i = 0 to 14
        if registers<i> == '1' then
            if i == n && wback && i != LowestSetBit(registers) then
                MemS[address,4] = bits(32) UNKNOWN;
            else
                MemS[address,4] = R[i];
                address = address + 4;
    if registers<15> == '1' then
        MemS[address,4] = PCStoreValue();
    if wback then R[n] = R[n] + 4*BitCount(registers);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

Encoding for the T1 variant

```
STR{<c>}{<q> <Rt>, [<Rn> {, #{+}<imm>}]}
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm5:'00', 32);
constant index = TRUE;  constant add = TRUE;  constant wback = 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							

Encoding for the T2 variant

```
STR{<c>}{<q> <Rt>, [SP{, #{+}<imm>}]}
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = 13;  constant imm32 = ZeroExtend(imm8:'00', 32);
constant index = TRUE;  constant add = TRUE;  constant wback = 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													
Rn																															

Encoding for the T3 variant

```
STR{<c>}.W <Rt>, [<Rn> {, #{+}<imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1 or T2)

STR{<c>}{<q> <Rt>, [<Rn> {, #{+}<imm>}]}
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt);  constant n = UInt(Rn);  constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
if t == 15 then UNPREDICTABLE;
```

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	1	0	0	!= 1111			Rt			1	P	U	W	imm8									
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";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

## 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 encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.

For encoding T1, T2, T3 and T4: is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For encoding A1: 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 encoding T1, T3 and T4: 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	+

+

Specifies the offset is added to the base register.

<imm> For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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 encoding T4: 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.

Alias Conditions

Alias	Of variant Is preferred when	
<a href="#">PUSH (single register)</a>	A1 (pre-indexed)	P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '000000000100'
<a href="#">PUSH (single register)</a>	T4 (pre-indexed)	Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'

Operation

```
if CurrentInstrSet() == InstrSet\_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    MemU[address,4] = if t == 15 then PCStoreValue() else R[t];
    if wback then R[n] = offset_addr;
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    MemU[address,4] = R[t];
    if wback then R[n] = offset_addr;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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																															

#### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

### 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		

Encoding for the T1 variant

```
STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	1	0	0	!= 1111				Rt				0	0	0	0	0	0	imm2				Rm			
Rn																																	

Encoding for the T2 variant

```
STR{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STR{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rt>	For encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.  For encoding T1 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For encoding A1: 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 encoding T1 and T2: 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	+						
+	Specifies the index register is added to the base register.						
<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*.
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    bits(32) data;
    if t == 15 then // Only possible for encoding A1
        data = PCStoreValue();
    else
        data = R[t];
    MemU[address,4] = data;
    if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

### Encoding for the T1 variant

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm5, 32);
constant index = TRUE; constant add = TRUE; constant wback = 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	!= 1111			Rt			imm12													

Rn

### Encoding for the T2 variant

```
STRB{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE; constant add = TRUE; constant wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

## 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									

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";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For encoding A1: 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 encoding T1, T2 and T3: 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	+						
+	Specifies the offset is added to the base register.						
<imm>	For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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.

## Operation

```
if CurrentInstrSet() == InstrSet\_A32 then
  if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed() then
      EncodingSpecificOperations();
      constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      constant address = if index then offset_addr else R[n];
      MemU[address,1] = R[t]<7:0>;
      if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

## 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

Encoding for the T1 variant

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	0	!= 1111			Rt			0 0 0 0 0 0							imm2			Rm			
Rn																															

Encoding for the T2 variant

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant index = TRUE;  constant add = TRUE;  constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: 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 encoding T1 and T2: 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>
  - <shift>
  - <imm>
- Specifies the index register is added to the base register.
- Is the general-purpose index register, encoded in the "Rm" field.
- The shift to apply to the value read from <Rm>. If absent, no shift is applied. Otherwise, see *Shifts applied to a register*.
- 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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    MemU[address,1] = R[t]<7:0>;
    if wback then R[n] = offset_addr;

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.



# 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																															

### Encoding for the A1 variant

STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1');
constant register_form = FALSE; constant imm32 = ZeroExtend(imm12, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

## 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																															

Encoding for the A2 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm); constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE; constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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.

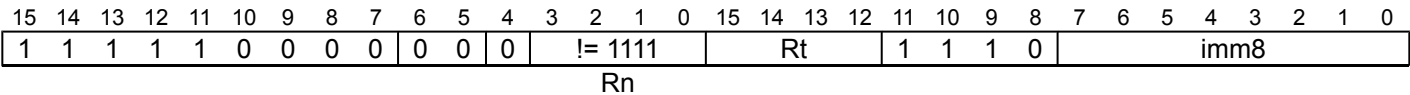
If `n == t`, then one of the following behaviors must occur:

- 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



Encoding for the T1 variant

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #(+)<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = FALSE; constant add = TRUE;
constant register_form = FALSE; constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.  
For encoding A2 and T1: 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 encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

For encoding A2: 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>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.
- <imm>For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
if PSTATE.EL == EL2 then UNPREDICTABLE;                                // Hyp mode
constant offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
constant address = if postindex then R[n] else offset_addr;
MemU_unpriv[address,1] = R[t]<7:0>;
if postindex then R[n] = offset_addr;
```

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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

## 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 UNPREDICTABLE;
constant t = UInt(Rt); constant t2 = t + 1; constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
if t2 == 15 then UNPREDICTABLE;
```

## 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

15	14	13	12	11	10	9	8	7	6	5	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	0	!= 1111			Rt				Rt2				imm8								
Rn																															

### 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";
constant t = UInt(Rt); constant t2 = UInt(Rt2); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rt>	For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.  For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.						
<Rt2>	For encoding A1: is the second general-purpose register to be transferred. This register must be <R(t+1)>.  For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.						
<Rn>	For encoding A1: 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 encoding T1: 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" data-bbox="240 869 384 960"> <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 encoding A1: 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 encoding T1: 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();
    constant_offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant_address = if index then offset_addr else R[n];
    if IsAligned(address, 8) then
        bits(64) data;
        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];
        MemA[address, 8] = data;
    else
        MemA[address, 4] = R[t];
        MemA[address+4, 4] = R[t2];
    if wback then R[n] = offset_addr;

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### 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 UNPREDICTABLE;
constant t = UInt(Rt); constant t2 = t + 1;
constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1');
constant wback = (P == '0') || (W == '1');
if P == '0' && W == '1' then UNPREDICTABLE;
if t2 == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t || n == t2) then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<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": <table border="1" data-bbox="240 965 384 1059"> <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	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    constant address = if index then offset_addr else R[n];
    if IsAligned(address, 8) then
        bits(64) data;
        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];
        MemA[address, 8] = data;
    else
        MemA[address, 4] = R[t];
        MemA[address+4, 4] = R[t2];
    if wback then R[n] = offset_addr;

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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																															

### Encoding for the A1 variant

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]

### Decode for this encoding

```
constant d = UInt(Rd); constant t = UInt(Rt); constant n = UInt(Rn);
constant imm32 = Zeros(32); // Zero offset
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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							

### Encoding for the T1 variant

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]

### Decode for this encoding

```
constant d = UInt(Rd); constant t = UInt(Rt); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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 encoding A1: the immediate offset added to the value of <Rn> to calculate the address. <imm> can only be 0 or omitted. For encoding T1: 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();
    constant address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address, 4) then
        MemA[address, 4] = R[t];
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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();
    constant address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t]<7:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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
!= 1111				0	0	0	1	1	0	1	0	Rn				Rd				(1)	(1)	1	1	1	0	0	1	Rt			
cond																															

### Encoding for the A1 variant

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

### Decode for this encoding

```
constant t = UInt(Rt);  constant t2 = t + 1;
constant d = UInt(Rd);  constant n = UInt(Rn);
if d == 15 || Rt<0> == '1' || t2 == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

## 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			

## Encoding for the T1 variant

```
STREXD{<c>}{<q>}<Rd>, <Rt>, <Rt2>, [<Rn>]
```

## Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant t2 = UInt(Rt2);  constant n = UInt(Rn);
if d == 15 || t == 15 || t2 == 15 || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if d == n || d == t || d == t2 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	<p>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:</p> <p><b>0</b></p> <p>If the operation updates memory.</p> <p><b>1</b></p> <p>If the operation fails to update memory.</p> <p>&lt;Rd&gt; must not be the same as &lt;Rn&gt;, &lt;Rt&gt;, or &lt;Rt2&gt;.</p>
<Rt>	<p>For encoding A1: is the first general-purpose register to be transferred, encoded in the "Rt" field. &lt;Rt&gt; must be even-numbered and not R14.</p> <p>For encoding T1: is the first general-purpose register to be transferred, encoded in the "Rt" field.</p>
<Rt2>	<p>For encoding A1: is the second general-purpose register to be transferred. &lt;Rt2&gt; must be &lt;R(t+1)&gt;.</p> <p>For encoding T1: is the second general-purpose register to be transferred, encoded in the "Rt2" field.</p>
<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();
    constant address = R[n];

    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    constant value = if BigEndian(AccessType\_GPR) then R[t]:R[t2] else R[t2]:R[t];

    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

### Decode for this encoding

```
constant d = UInt(Rd);  constant t = UInt(Rt);  constant n = UInt(Rn);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE;
if d == n || d == t then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <b>0</b> If the operation updates memory. <b>1</b> 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();
    constant address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0', 32);
    else
        R[d] = ZeroExtend('1', 32);
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
if t == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

### Encoding for the T1 variant

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm5:'0', 32);
constant index = TRUE; constant add = TRUE; constant wback = 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													

Rn

### Encoding for the T2 variant

```
STRH{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm12, 32);
constant index = TRUE; constant add = TRUE; constant wback = FALSE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

## 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									

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";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8, 32);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (W == '1');
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || (wback && n == t) then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .						
<q>	See <a href="#">Standard assembler syntax fields</a> .						
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.						
<Rn>	For encoding A1: 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 encoding A1, T1, T2, T3: 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	+						
+	Specifies the offset is added to the base register.						
<imm>	For encoding A1: 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 encoding T1: 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 encoding T2: 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 encoding T3: 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.

## Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    constant address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;
  else
    if ConditionPassed\(\) then
      EncodingSpecificOperations();
      constant offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
      constant address = if index then offset_addr else R[n];
      MemU[address,2] = R[t]<15:0>;
      if wback then R[n] = offset_addr;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

### 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";
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = (P == '1'); constant add = (U == '1'); constant wback = (P == '0') || (W == '1');
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
if t == 15 || m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
```

## 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		

### Encoding for the T1 variant

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = TRUE; constant add = TRUE; constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, 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	1	0	!= 1111			Rt			0			0	0	0	0	imm2			Rm			
Rn																															

### Encoding for the T2 variant

```
STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

```
STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

### Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm);
constant index = TRUE; constant add = TRUE; constant wback = FALSE;
constant (shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || m == 15 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	For encoding A1: 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 encoding T1 and T2: 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	+

- +
- Specifies the index register is added to the base register.
- <Rm>
- Is the general-purpose index register, encoded in the "Rm" field.
- <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();
    constant offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if index then offset_addr else R[n];
    MemU[address,2] = R[t]<15:0>;
    if wback then R[n] = offset_addr;

```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #(<+/->)<imm>}

### Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant postindex = TRUE;
constant add = (U == '1');
constant register_form = FALSE;  constant imm32 = ZeroExtend(imm4H:imm4L, 32);
constant m = integer UNKNOWN;
if t == 15 || n == 15 || n == t then UNPREDICTABLE;
```

## 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																															

Encoding for the A2 variant

```
STRHT{<c>}{<q>}<Rt>, [<Rn>], {+/-}<Rm>
```

Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant m = UInt(Rm); constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE;
constant bits(32) imm32 = bits(32) UNKNOWN;
if t == 15 || n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

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.

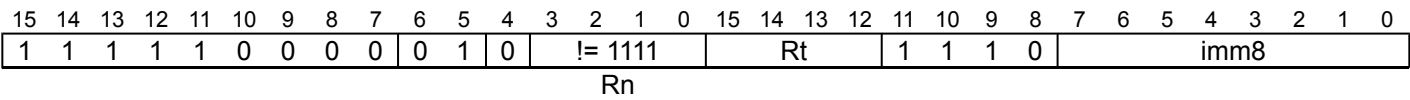
If `n == t`, then one of the following behaviors must occur:

- 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



Encoding for the T1 variant

```
STRHT{<c>}{<q>}<Rt>, [<Rn> {, #(+)<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = FALSE; constant add = TRUE;
constant register_form = FALSE; constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1: specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

For encoding A2: 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 offset is added to the base register.
- <imm>For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
constant offset = if register_form then R[m] else imm32;
constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
constant address = if postindex then R[n] else offset_addr;
MemU_unpriv[address,2] = R[t]<15:0>;
if postindex then R[n] = offset_addr;
```

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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																															

### Encoding for the A1 variant

STRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}

### Decode for this encoding

```
constant t = UInt(Rt); constant n = UInt(Rn); constant postindex = TRUE;
constant add = (U == '1');
constant register_form = FALSE; constant imm32 = ZeroExtend(imm12, 32);
constant m = integer UNKNOWN; constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
if n == 15 || n == t then UNPREDICTABLE;
```

## 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																															

Encoding for the A2 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Decode for this encoding

```
constant t = UInt(Rt);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant postindex = TRUE;
constant add = (U == '1');
constant register_form = TRUE;  constant (shift_t, shift_n) = DecodeImmShift(stype, imm5);
constant bits(32) imm32 = bits(32) UNKNOWN;
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

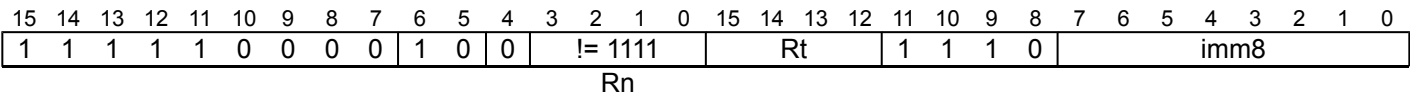
If `n == t`, then one of the following behaviors must occur:

- 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



Encoding for the T1 variant

```
STRT{<c>}{<q>} <Rt>, [<Rn> {, #{+}<imm>}]
```

Decode for this encoding

```
if Rn == '1111' then UNDEFINED;
constant t = UInt(Rt);  constant n = UInt(Rn);  constant postindex = FALSE;  constant add = TRUE;
constant register_form = FALSE;  constant imm32 = ZeroExtend(imm8, 32);
constant m = integer UNKNOWN;  constant shift_n = integer UNKNOWN;
constant SRTYPE shift_t = SRTYPE UNKNOWN;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 encoding A1 and A2: is the general-purpose register to be transferred, encoded in the "Rt" field. The PC can be used, but this is deprecated.						
	For encoding T1: 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 encoding A1: 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 encoding A2: 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	+						
<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.						
<imm>	For encoding A1: 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 encoding T1: is an optional 8-bit unsigned immediate byte offset, in the range 0 to 255, defaulting to 0 and encoded in the "imm8" field.						

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL2 then UNPREDICTABLE; // Hyp mode
    constant offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    constant offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    constant address = if postindex then R[n] else offset_addr;
    bits(32) data;
    if t == 15 then // Only possible for encodings A1 and A2
        data = PCStoreValue();
    else
        data = R[t];
    MemU_unpriv[address,4] = data;
    if postindex then R[n] = offset_addr;

```

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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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																																

### 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							

### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rd>	For encoding A2: 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 <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a> .  For encoding T2: is the general-purpose destination register, encoded in the "Rd" field.
<label>	For encoding A2: 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 <a href="#">Modified immediate constants in A32 instructions</a> .  For encoding T2: 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.

<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.

## Operation

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

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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																															

### 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";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = (S == '1');
constant imm32 = 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		

### Encoding for the T1 variant

```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // (Inside IT block)
```

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

### Decode for this encoding

```
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = !InITBlock();
constant imm32 = ZeroExtend(imm3, 32);
```

## T2

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

### Encoding for the T2 variant

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

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

```
constant d = UInt(Rdn);  constant n = UInt(Rdn);  constant setflags = !InITBlock();
constant imm32 = ZeroExtend(imm8, 32);
```

## 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									

Rn

### Encoding for the SUB variant

Applies when (S == 0)

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

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

### Encoding for the SUBS variant

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

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

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

### Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant setflags = (S == '1');
constant imm32 = T32ExpandImm(i:imm3:imm8);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
```

## 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								

Rn

Encoding for the T4 variant

```
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";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
constant d = UInt(Rd); constant n = UInt(Rn); constant setflags = FALSE;
constant imm32 = ZeroExtend(i:imm3:imm8, 32);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 then UNPREDICTABLE;
```

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 for the T5 variant

Applies when (!(Rn == 1110 && imm8 == 00000000))

```
SUBS{<c>}{<q>} PC, LR, #<imm8>
```

Decode for this encoding

```
if Rn == '1110' && IsZero(imm8) then SEE "ERET";
constant d = 15; constant n = UInt(Rn); constant setflags = TRUE;
constant imm32 = ZeroExtend(imm8, 32);
if n != 14 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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*.
- <Rdn> Is the general-purpose source and destination register, encoded in the "Rdn" field.
- <imm8> For encoding T2: is a 8-bit unsigned immediate, in the range 0 to 255, encoded in the "imm8" field.  
For encoding T5: 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*.
- <Rd> For encoding A1: 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 *PSTATE* from SPSR\_<current\_mode>. Arm deprecates use of this instruction unless <Rn> is the LR.For encoding T1, T3 and T4: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.
- <Rn> For encoding A1 and T4: 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 encoding T1: is the general-purpose source register, encoded in the "Rn" field.  
For encoding T3: is the general-purpose source register, encoded in the "Rn" field. If the SP is used, see *SUB (SP minus immediate)*.
- <imm3> Is a 3-bit unsigned immediate, in the range 0 to 7, encoded in the "imm3" field.

<imm12> Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.

<const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

For encoding T3: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

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();
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(imm32), '1');
    if d == 15 then
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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												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)";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = 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

Encoding for the T1 variant

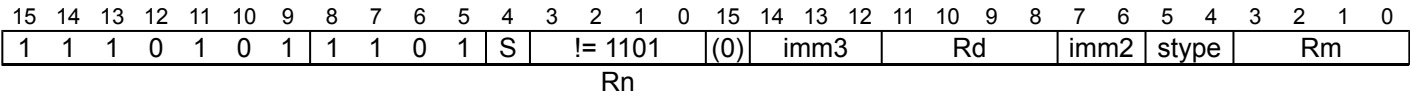
```
SUB<c>{<q> <Rd>, <Rn>, <Rm> // (Inside IT block)

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

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = !InITBlock();
constant (shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2



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

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

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 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

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

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || n == 15 || m == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .										
<Rd>	<p>For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as &lt;Rn&gt;. Arm deprecates using the PC as the destination register, but if the PC is used:</p> <ul style="list-style-type: none"> <li>For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li> <li>For the SUBS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;.</li> </ul> <p>For encoding T1 and T2: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as &lt;Rn&gt;.</p>										
<Rn>	<p>For encoding A1: 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 <a href="#">SUB (SP minus register)</a>.</p> <p>For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.</p> <p>For encoding T2: is the first general-purpose source register, encoded in the "Rn" field. If the SP is used, see <a href="#">SUB (SP minus register)</a>.</p>										
<Rm>	<p>For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.</p> <p>For encoding T1 and T2: is the second general-purpose source register, encoded in the "Rm" field.</p>										
<shift>	<p>Is the type of shift to be applied to the second source register, encoded in "stype":</p> <table> <tr> <th>stype</th><th>&lt;shift&gt;</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>	<p>For encoding A1: is the shift amount, in the range 1 to 31 (when &lt;shift&gt; = LSL or ROR) or 1 to 32 (when &lt;shift&gt; = LSR or ASR) encoded in the "imm5" field as &lt;amount&gt; modulo 32.</p> <p>For encoding T2: is the shift amount, in the range 1 to 31 (when &lt;shift&gt; = LSL or ROR) or 1 to 32 (when &lt;shift&gt; = LSR or ASR), encoded in the "imm3:imm2" field as &lt;amount&gt; modulo 32.</p>										

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), '1');
    if d == 15 then // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;

```

Operational information

- If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant s = UInt(Rs);
constant setflags = (S == '1');  constant shift_t = DecodeRegShift(stype);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[n], NOT(shifted), '1');
    R[d] = result;
    if setflags then
        PSTATE.<N,Z,C,V> = nzcvc;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### 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

```
constant d = UInt(Rd);  constant setflags = (S == '1');  constant imm32 = 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						

### Encoding for the T1 variant

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

### Decode for this encoding

```
constant d = 13;  constant setflags = FALSE;  constant imm32 = ZeroExtend(imm7:'00', 32);
```

## 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							

### Encoding for the SUB variant

Applies when (S == 0)

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

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

## 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)";
constant d = UInt(Rd); constant setflags = (S == '1'); constant imm32 = T32ExpandImm(i:imm3:imm8);
if d == 15 && !setflags then UNPREDICTABLE;
```

## 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														

## Encoding for the T3 variant

```
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

```
constant d = UInt(Rd); constant setflags = FALSE; constant imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
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.
<Rd>	For encoding A1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP. If the PC is used: <ul style="list-style-type: none"> <li>For the SUB variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see <a href="#">Pseudocode description of operations on the AArch32 general-purpose registers and the PC</a>.</li> <li>For the SUBS variant, the instruction performs an exception return, that restores <a href="#">PSTATE</a> from SPSR_&lt;current_mode&gt;. Arm deprecates use of this instruction unless &lt;Rn&gt; is the LR.</li> </ul> For encoding T2 and T3: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
<imm12>	Is a 12-bit unsigned immediate, in the range 0 to 4095, encoded in the "i:imm3:imm8" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T2: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (result, nzcvc) = AddWithCarry(R[13], NOT(imm32), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### 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

```
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = (S == '1');  
constant (shift_t, shift_n) = 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			

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

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

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>, RRX

Encoding for the SUBS, shift or rotate by value variant

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

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

Decode for all variants of this encoding

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
constant d = UInt(Rd); constant m = UInt(Rm); constant setflags = (S == '1');
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if (d == 15 && !setflags) || m == 15 then UNPREDICTABLE;
```

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 encoding A1: 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>.For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the SP.
- <Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    constant (result, nzcvc) = AddWithCarry(R[13], NOT(shifted), '1');
    if d == 15 then          // Can only occur for A32 encoding
        if setflags then
            ALUExceptionReturn(result);
        else
            ALUWritePC(result);
    else
        R[d] = result;
        if setflags then
            PSTATE.<N,Z,C,V> = nzcvc;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 for the A1 variant

SVC{<c>}{<q>} {#}<imm>

## Decode for this encoding

```
constant imm32 = ZeroExtend(imm24, 32);
```

## 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							

## Encoding for the T1 variant

SVC{<c>}{<q>} {#}<imm>

## Decode for this encoding

```
constant imm32 = ZeroExtend(imm8, 32);
```

## Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <imm> For encoding A1: 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 encoding T1: 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>);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

31	30	29	28	27	26	25	24	23	22	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	!= 1111				Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond												Rn																			

Encoding for the A1 variant

SXTAB{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
Rn																															

Encoding for the T1 variant

SXTAB{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<7:0>, 32);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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												Rn																			

Encoding for the A1 variant

SXTAB16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB16";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				
Rn																															

Encoding for the T1 variant

SXTAB16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTB16";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d]<15:0> = R[n]<15:0> + SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + SignExtend(rotated<23:16>, 16);

```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	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	!= 1111				Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond												Rn																			

Encoding for the A1 variant

SXTAH{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTH";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	!= 1111				1	1	1	1	Rd				1	(0)	rotate	Rm				
Rn																															

Encoding for the T1 variant

SXTAH{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "SXTH";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = R[n] + SignExtend(rotated<15:0>, 32);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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	0	1	1	1	1	Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond																															

### Encoding for the A1 variant

SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

SXTB{<c>}{<q>} {<Rd>,} <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = 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				

### Encoding for the T2 variant

SXTB{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

SXTB{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<7:0>, 32);
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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																															

Encoding for the A1 variant

```
SXTB16{<c>}{<q>} {<Rd>}, {<Rm> {, ROR #<amount>}}
```

Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				

Encoding for the T1 variant

```
SXTB16{<c>}{<q>} {<Rd>}, {<Rm> {, ROR #<amount>}}
```

Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d]<15:0> = SignExtend(rotated<7:0>, 16);
    R[d]<31:16> = SignExtend(rotated<23:16>, 16);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

SXTH{<c>}{<q>} {<Rd>,} <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = 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				

### Encoding for the T2 variant

SXTH{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

SXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = SignExtend(rotated<15:0>, 32);
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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

15	14	13	12	11	10	9	8	7	6	5	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				(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	H	Rm			

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

```
constant n = UInt(Rn); constant m = UInt(Rm); constant is_tbh = (H == '1');
// Armv8-A removes UNPREDICTABLE for R13
if m == 15 then UNPREDICTABLE;
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

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();
    integer halfwords;
    if is_tbh then
        halfwords = UInt(MemU[R[n]+LSL(R[m],1), 2]);
    else
        halfwords = UInt(MemU[R[n]+R[m], 1]);
    BranchWritePC(PC32 + 2*halfwords, BranchType_INDIR);
```

## 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

31	30	29	28	27	26	25	24	23	22	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	1	Rn				(0)	(0)	(0)	(0)	imm12											
cond																															

#### Encoding for the A1 variant

TEQ{<c>}{<q>} <Rn>, #<const>

#### Decode for this encoding

```
constant n = UInt(Rn); constant a32 = TRUE;
constant bits(12) imm = 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	1	Rn				0	imm3				1	1	1	1	imm8							

#### Encoding for the T1 variant

TEQ{<c>}{<q>} <Rn>, #<const>

#### Decode for this encoding

```
constant n = UInt(Rn); constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rn>	For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
<const>	For encoding A1: an immediate value. See <a href="#">Modified immediate constants in A32 instructions</a> for the range of values. For encoding T1: an immediate value. See <a href="#">Modified immediate constants in T32 instructions</a> for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm\_C(imm, PSTATE.C)
                               else T32ExpandImm\_C(imm, PSTATE.C));
    constant result = R[n] EOR imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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																															

#### 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

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = 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			

#### 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

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1: is the first general-purpose source register, encoded in the "Rn" field.
- <Rm>For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1: 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 encoding A1: 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 encoding T1: 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();
    constant (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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

31	30	29	28	27	26	25	24	23	22	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)	Rs				0	stype	1	Rm				
cond																															

### Encoding for the A1 variant

TEQ{<c>}{<q>}<Rn>, <Rm>, <type> <Rs>

### Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] EOR shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

### Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 ( [A1](#) ) and T32 ( [T1](#) ).

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																															

Encoding for the A1 variant

TSB{<c>}{<q>} CSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP();
if cond != '1110' then UNPREDICTABLE; // 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

Encoding for the T1 variant

TSB{<c>}{<q>} CSYNC

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_TRF) then ExecuteAsNOP();
if InITBlock() then UNPREDICTABLE;
```

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\(\);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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	1	Rn				(0)	(0)	(0)	(0)	imm12											
cond																															

### Encoding for the A1 variant

TST{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn); constant a32 = TRUE;
constant bits(12) imm = 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	1	Rn				0	imm3				1	1	1	1	imm8							

### Encoding for the T1 variant

TST{<c>}{<q>} <Rn>, #<const>

### Decode for this encoding

```
constant n = UInt(Rn); constant a32 = FALSE;
constant bits(12) imm = i:imm3:imm8;
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1: is the general-purpose source register, encoded in the "Rn" field.
- <const> For encoding A1: an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.  
For encoding T1: an immediate value. See [Modified immediate constants in T32 instructions](#) for the range of values.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant (imm32, carry) = (if a32 then A32ExpandImm\_C(imm, PSTATE.C)
                               else T32ExpandImm\_C(imm, PSTATE.C));
    constant result = R[n] AND imm32;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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																															

#### 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

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = 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	

#### Encoding for the T1 variant

```
TST{<c>}{<q>} <Rn>, <Rm>
```

#### Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = (SRTType_LSL, 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			

#### 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>}.W <Rn>, <Rm> // (<Rn>, <Rm> can be represented in T1)

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

Decode for all variants of this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm);
constant (shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if n == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.  
For encoding T1 and T2: is the first general-purpose source register, encoded in the "Rn" field.
- <Rm> For encoding A1: is the second general-purpose source register, encoded in the "Rm" field. The PC can be used, but this is deprecated.  
For encoding T1 and T2: 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 encoding A1: 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 encoding T2: 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();
    constant (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# 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

31	30	29	28	27	26	25	24	23	22	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)	Rs				0	stype	1	Rm				
cond																															

## Encoding for the A1 variant

TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

## Decode for this encoding

```
constant n = UInt(Rn); constant m = UInt(Rm); constant s = UInt(Rs);
constant shift_t = DecodeRegShift(stype);
if n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

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();
    constant shift_n = UInt(R[s]<7:0>);
    constant (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    constant result = R[n] AND shifted;
    PSTATE.N = result<31>;
    PSTATE.Z = IsZeroBit(result);
    PSTATE.C = carry;
    // PSTATE.V unchanged
```

## Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

UADD16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UADD16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    constant sum2 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

UADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

UADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    constant sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    constant sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    constant sum4 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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																															

### Encoding for the A1 variant

UASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    constant sum  = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																															

#### Encoding for the A1 variant

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

#### Decode for this encoding

```
constant d = UInt(Rd); constant n = UInt(Rn);
constant integer lsbit = UInt(lsb);
constant integer widthminus1 = UInt(widthm1);
constant integer msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

#### 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			

#### Encoding for the T1 variant

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

#### Decode for this encoding

```
constant d = UInt(Rd); constant n = UInt(Rn);
constant integer lsbit = UInt(imm3:imm2);
constant integer widthminus1 = UInt(widthm1);
constant integer msbit = lsbit + widthminus1;
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
if msbit > 31 then UNPREDICTABLE;
```

#### 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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 encoding A1: is the bit number of the least significant bit in the field, in the range 0 to 31, encoded in the "lsb" field. For encoding T1: 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(R[n]<msbit:lsbit>, 32);
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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 for the A1 variant

```
UDF{<c>}{<q>} {#}<imm>
```

### Decode for this encoding

```
constant imm32 = ZeroExtend(imm12:imm4, 32);
// 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							

### Encoding for the T1 variant

```
UDF{<c>}{<q>} {#}<imm>
```

### Decode for this encoding

```
constant imm32 = ZeroExtend(imm8, 32);
// 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											

### Encoding for the T2 variant

```
UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)

UDF{<c>}{<q>} {#}<imm>
```

### Decode for this encoding

```
constant imm32 = ZeroExtend(imm4:imm12, 32);
// imm32 is for assembly and disassembly only, and is ignored by hardware.
```

## Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. <c> must be AL or omitted.
- For encoding T1 and T2: see *Standard assembler syntax fields*. Arm deprecates using any <c> value other than AL.

<q> See *Standard assembler syntax fields*.

<imm> For encoding A1: 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 encoding T1: 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 encoding T2: 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;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												Ra																			

### Encoding for the A1 variant

UDIV{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

### 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			
Ra																															

### Encoding for the T1 variant

UDIV{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE;
```

### 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 <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 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();
    constant integer dividend = UInt(R[n]);
    constant integer divisor  = UInt(R[m]);
    integer result;
    if divisor == 0 then
        result = 0;
    else
        result = dividend DIV divisor;
    R[d] = result<31:0>;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	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	0	1	Rm			
cond																															

### Encoding for the A1 variant

UHADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

UHADD16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    constant sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0> = sum1<16:1>;
    R[d]<31:16> = sum2<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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)	1	0	0	1	Rm			
cond																															

### Encoding for the A1 variant

UHADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

UHADD8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    constant sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    constant sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    constant sum4 = 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>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

UHASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UHASX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    constant sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>    = diff<16:1>;
    R[d]<31:16>   = sum<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

UHSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UHSAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    constant diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>    = sum<16:1>;
    R[d]<31:16>   = diff<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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

31	30	29	28	27	26	25	24	23	22	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	1	1	Rm			
cond																															

### Encoding for the A1 variant

UHSUB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

UHSUB16{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    constant diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0> = diff1<16:1>;
    R[d]<31:16> = diff2<16:1>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	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)	1	1	1	1	Rm			
cond																															

### Encoding for the A1 variant

UHSUB8{<c>}{<q>}{<Rd>,}<Rn>,<Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

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

### Encoding for the T1 variant

UHSUB8{<c>}{<q>}{<Rd>,}<Rn>,<Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    constant diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    constant diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    constant diff4 = 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>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

### 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	1	1	0	Rm			

### Encoding for the T1 variant

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

### 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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<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();
    constant result = UInt(R[n]) * UInt(R[m]) + UInt(R[dHi]) + UInt(R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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																															

### 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant result = 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(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.



# 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																															

### 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

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = (S == '1');
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
if dHi == dLo then UNPREDICTABLE;
```

## 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			

### Encoding for the T1 variant

UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

### Decode for this encoding

```
constant dLo = UInt(RdLo);  constant dHi = UInt(RdHi);  constant n = UInt(Rn);
constant m = UInt(Rm);  constant setflags = FALSE;
if dLo == 15 || dHi == 15 || n == 15 || m == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if dHi == dLo then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant result = 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(result<63:0>);
        // PSTATE.C, PSTATE.V unchanged
```

Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

# 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																															

### Encoding for the A1 variant

UQADD16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UQADD16{<c>}{<q>} {<Rd>,<Rn>,<Rm>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    constant sum2 = UInt(R[n]<31:16>) + UInt(R[m]<31:16>);
    R[d]<15:0> = UnsignedSat(sum1, 16);
    R[d]<31:16> = UnsignedSat(sum2, 16);
```



# 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																															

### Encoding for the A1 variant

UQADD8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UQADD8{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    constant sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    constant sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    constant sum4 = UInt(R[n]<31:24>) + UInt(R[m]<31:24>);
    R[d]<7:0>    = UnsignedSat(sum1, 8);
    R[d]<15:8>   = UnsignedSat(sum2, 8);
    R[d]<23:16>  = UnsignedSat(sum3, 8);
    R[d]<31:24>  = UnsignedSat(sum4, 8);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UQASX{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    constant sum  = UInt(R[n]<31:16>) + UInt(R[m]<15:0>);
    R[d]<15:0>    = UnsignedSat(diff, 16);
    R[d]<31:16>   = UnsignedSat(sum, 16);
```





# 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																															

### Encoding for the A1 variant

UQSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UQSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    constant diff = UInt(R[n]<31:16>) - UInt(R[m]<15:0>);
    R[d]<15:0>    = UnsignedSat(sum, 16);
    R[d]<31:16>   = UnsignedSat(diff, 16);
```



# 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																															

### Encoding for the A1 variant

UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

UQSUB16{<c>}{<q>} {<Rd>,} <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    constant diff2 = UInt(R[n]<31:16>) - UInt(R[m]<31:16>);
    R[d]<15:0> = UnsignedSat(diff1, 16);
    R[d]<31:16> = UnsignedSat(diff2, 16);
```



# 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

31	30	29	28	27	26	25	24	23	22	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	1	1	1	Rm			
cond																															

### Encoding for the A1 variant

UQSUB8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	Rm			

### Encoding for the T1 variant

UQSUB8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    constant diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    constant diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    constant diff4 = UInt(R[n]<31:24>) - UInt(R[m]<31:24>);
    R[d]<7:0>    = UnsignedSat(diff1, 8);
    R[d]<15:8>   = UnsignedSat(diff2, 8);
    R[d]<23:16>  = UnsignedSat(diff3, 8);
    R[d]<31:24>  = UnsignedSat(diff4, 8);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

USAD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

USAD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant absdiff1 = Abs(UInt(R[n]<7:0>) - UInt(R[m]<7:0>));
    constant absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
    constant absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    constant absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    constant result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```





# 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 for the A1 variant

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			
Ra																															

### Encoding for the T1 variant

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

### Decode for this encoding

```
if Ra == '1111' then SEE "USAD8";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);  constant a = UInt(Ra);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant absdiff1 = Abs(UInt(R[n]<7:0>) - UInt(R[m]<7:0>));
    constant absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
    constant absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    constant absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    constant result = UInt(R[a]) + absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### 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

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm);
constant (shift_t, shift_n) = DecodeImmShift(sh:'0', imm5);
if d == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	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) == '00000' then SEE "USAT16";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm);
constant (shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
```

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.
<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 encoding A1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm5" field. For encoding A1: is the shift amount, in the range 1 to 32 encoded in the "imm5" field as <amount> modulo 32. For encoding T1: is the optional shift amount, in the range 0 to 31, defaulting to 0 and encoded in the "imm3:imm2" field. For encoding T1: is the shift amount, in the range 1 to 31 encoded in the "imm3:imm2" field as <amount>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant operand = Shift(R[n], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
    constant (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
    R[d] = ZeroExtend(result, 32);
    if sat then
        PSTATE.Q = '1';
```

# 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																															

### Encoding for the A1 variant

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm);
if d == 15 || n == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	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			

### Encoding for the T1 variant

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant saturate_to = UInt(sat_imm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 then UNPREDICTABLE;
```

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();
    constant (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    constant (result2, sat2) = UnsignedSatQ(SInt(R[n]<31:16>), saturate_to);
    R[d]<15:0> = ZeroExtend(result1, 16);
    R[d]<31:16> = ZeroExtend(result2, 16);
    if sat1 || sat2 then
        PSTATE.Q = '1';
```



# 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																															

### Encoding for the A1 variant

USAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

USAX{<c>}{<q>} {<Rd>, } <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant sum  = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    constant diff = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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																															

### Encoding for the A1 variant

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

### Encoding for the T1 variant

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    constant diff2 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Encoding for the A1 variant

USUB8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Encoding for the T1 variant

USUB8{<c>}{<q>}{<Rd>, } <Rn>, <Rm>

Decode for this encoding

```
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || n == 15 || m == 15 then UNPREDICTABLE;
```

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();
    constant diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    constant diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    constant diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    constant diff4 = 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';
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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	1	0	!= 1111				Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond												Rn																			

### Encoding for the A1 variant

UXTAB{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

### Decode for this encoding

```
if Rn == '1111' then SEE "UXTB";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				
Rn																															

### Encoding for the T1 variant

UXTAB{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

### Decode for this encoding

```
if Rn == '1111' then SEE "UXTB";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<7:0>, 32);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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												Rn																			

Encoding for the A1 variant

UXTAB16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB16";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				
Rn																															

Encoding for the T1 variant

UXTAB16{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

Decode for this encoding

```
if Rn == '1111' then SEE "UXTB16";
constant d = UInt(Rd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d]<15:0> = R[n]<15:0> + ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = R[n]<31:16> + ZeroExtend(rotated<23:16>, 16);

```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												Rn																			

### Encoding for the A1 variant

UXTAH{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

### Decode for this encoding

```
if Rn == '1111' then SEE "UXTH";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				
Rn																															

### Encoding for the T1 variant

UXTAH{<c>}{<q>} {<Rd>}, {<Rn>, <Rm> {, ROR #<amount>}}

### Decode for this encoding

```
if Rn == '1111' then SEE "UXTH";
constant d = UInt(Rd); constant n = UInt(Rn); constant m = UInt(Rm);
constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = R[n] + ZeroExtend(rotated<15:0>, 32);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	1	0	1	1	1	1	Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond																															

### Encoding for the A1 variant

UXTB{<c>}{<q>} {<Rd>, } <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

UXTB{<c>}{<q>} {<Rd>, } <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = 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				

### Encoding for the T2 variant

UXTB{<c>}.W {<Rd>, } <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTB{<c>}{<q>} {<Rd>, } <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<7:0>, 32);
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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																															

Encoding for the A1 variant

```
UXTB16{<c>}{<q>} {<Rd>,<Rm> {, ROR #<amount>}}
```

Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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				

Encoding for the T1 variant

```
UXTB16{<c>}{<q>} {<Rd>,<Rm> {, ROR #<amount>}}
```

Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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 encoding A1: is the general-purpose source register, encoded in the "Rm" field.  
For encoding T1: is the second general-purpose source register, encoded in the "Rm" field.
- <amount> Is the rotate amount, encoded in “rotate”:

rotate	<amount>
00	(omitted)
01	8
10	16
11	24

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d]<15:0> = ZeroExtend(rotated<7:0>, 16);
    R[d]<31:16> = ZeroExtend(rotated<23:16>, 16);
```

## Operational information

If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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	Rd				rotate	(0)	(0)	0	1	1	1	Rm					
cond																															

### Encoding for the A1 variant

UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
if d == 15 || m == 15 then UNPREDICTABLE;
```

## 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	

### Encoding for the T1 variant

UXTH{<c>}{<q>} {<Rd>,} <Rm>

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = 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	1	Rd				1	(0)	rotate	Rm			

### Encoding for the T2 variant

UXTH{<c>}.W {<Rd>,} <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTH{<c>}{<q>} {<Rd>,} <Rm> {, ROR #<amount>}

### Decode for this encoding

```
constant d = UInt(Rd);  constant m = UInt(Rm);  constant rotation = UInt(rotate:'000');
// Armv8-A removes UNPREDICTABLE for R13
if d == 15 || m == 15 then UNPREDICTABLE;
```

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	(omitted)
01	8
10	16
11	24

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    constant rotated = ROR(R[m], rotation);
    R[d] = ZeroExtend(rotated<15:0>, 32);
```

Operational information

- If CPSR.DIT is 1, this instruction has passed its condition execution check, and does not use R15 as either its source or destination:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



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																															

Encoding for the A1 variant

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

Encoding for the T1 variant

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

Encoding for the T2 variant

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);
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS.
        AArch32.CheckForWfxTrap(EL1, WfxType\_WFE);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch32.CheckForWfxTrap(EL2, WfxType\_WFE);
    if HaveEL(EL3) && PSTATE.M != M32\_Monitor then
        // Check for traps described by the Secure Monitor.
        AArch32.CheckForWfxTrap(EL3, WfxType\_WFE);
    WaitForEvent();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### Encoding for the A1 variant

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

### Encoding for the T1 variant

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

### Encoding for the T2 variant

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 !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);
    if PSTATE.EL == EL0 then
        // Check for traps described by the OS.
        AArch32.CheckForWfxTrap(EL1, WfxType\_WFI);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch32.CheckForWfxTrap(EL2, WfxType\_WFI);
    if HaveEL(EL3) && PSTATE.M != M32\_Monitor then
        // Check for traps described by the Secure Monitor.
        AArch32.CheckForWfxTrap(EL3, WfxType\_WFI);
    WaitForInterrupt();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	1
cond																															

### Encoding for the A1 variant

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

### Encoding for the T1 variant

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

### Encoding for the T2 variant

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();
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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, VFMA \(by scalar\)](#): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

[VFMA, VFMA \(vector\)](#): Vector Floating-point Multiply-Add Long to accumulator (vector).

[VFMS](#): Vector Fused Multiply Subtract.

[VFMS, VFMS \(by scalar\)](#): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

[VFMS, VFMS \(vector\)](#): 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.

[VRSORTE](#): Vector Reciprocal Square Root Estimate.

[VRSQRTS](#): Vector Reciprocal Square Root Step.

[VRSA](#): 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	0	1	M	0	Vm					

### Encoding for the A1 variant

AESD.<dt> <Qd>, <Qm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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	0	1	M	0		Vm			

### Encoding for the T1 variant

AESD.<dt> <Qd>, <Qm>

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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();
    constant op1 = Q[d>>1]; constant op2 = Q[m>>1];
    Q[d>>1] = AESInvSubBytes(AESInvShiftRows(op1 EOR op2));
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# AESE

AES single round encryption.  
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	1	1	0	0	M	0	Vm					

### Encoding for the A1 variant

AESE.<dt> <Qd>, <Qm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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	0	0	M	0		Vm			

### Encoding for the T1 variant

AESE.<dt> <Qd>, <Qm>

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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();
    constant op1 = Q[d>>1]; constant op2 = Q[m>>1];
    Q[d>>1] = AESSubBytes(AESShiftRows(op1 EOR op2));
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	1	1	1	1	M	0	Vm					

Encoding for the A1 variant

```
AESIMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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			

Encoding for the T1 variant

```
AESIMC.<dt> <Qd>, <Qm>
```

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
```

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();
    Q[d>>1] = AESInvMixColumns(Q[m>>1]);
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESMC

AES 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	1	1	1	0	M	0	Vm					

Encoding for the A1 variant

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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	0	M	0		Vm			

Encoding for the T1 variant

AESMC.<dt> <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AES) then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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();
     $Q[d \gg 1] = \text{AESMixColumns}(Q[m \gg 1]);$ 
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 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
!= 1111				1	1	0	P	U	D	W	1	Rn				Vd				1	0	1	1	imm8<7:1>				1			
cond																imm8<0>															

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";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

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	imm8<7:1>				1			
																imm8<0>															

## 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";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
!	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<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.



## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;

    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];
            address = address+4;
        else
            constant word1 = MemA[address,4];
            constant word2 = MemA[address+4,4];
            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;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	imm8<7:1>				1			
cond																imm8<0>															

### 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";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## 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	imm8<7:1>				1		imm8<0>	

## 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";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
!	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = 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[address,4] = D[d+r]<63:32>;
                MemA[address+4,4] = D[d+r]<31:0>;
            else
                MemA[address,4] = D[d+r]<31:0>;
                MemA[address+4,4] = D[d+r]<63:32>;

            address = address+8;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)

31	30	29	28	27	26	25	24	23	22	21	20	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				1	1	0	0	N	Q	M	0	Vm			

Encoding for the A1 variant

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	0	0	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

Encoding for the T1 variant

SHA1C.32 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    x = Q[d>>1];
    y = Q[n>>1]<31:0>; // Note: 32 bits wide
    constant w = Q[m>>1];
    for e = 0 to 3
        constant t = SHAchoose(x<63:32>, x<95:64>, x<127:96>);
        y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
        x<63:32> = ROL(x<63:32>, 30);
        <y, x> = ROL(y:x, 32);
    Q[d>>1] = x;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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					

Encoding for the A1 variant

SHA1H.32 <Qd>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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			

Encoding for the T1 variant

SHA1H.32 <Qd>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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(ROL(Q[m>>1]<31:0>, 30), 128);
```

Operational information

- If CPSR.DIT is 1:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



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			

Encoding for the A1 variant

SHA1M.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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					

Encoding for the T1 variant

SHA1M.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    x = Q[d>>1];
    y = Q[n>>1]<31:0>; // Note: 32 bits wide
    constant w = Q[m>>1];
    for e = 0 to 3
        constant t = SHAMajority(x<63:32>, x<95:64>, x<127:96>);
        y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
        x<63:32> = ROL(x<63:32>, 30);
        <y, x> = ROL(y:x, 32);
    Q[d>>1] = x;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SHA1P

SHA1 hash update (parity).  
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	0	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

### Encoding for the A1 variant

SHA1P.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	0	1	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

### Encoding for the T1 variant

SHA1P.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    x = Q[d>>1];
    y = Q[n>>1]<31:0>; // Note: 32 bits wide
    constant w = Q[m>>1];
    for e = 0 to 3
        constant t = SHAparity(x<63:32>, x<95:64>, x<127:96>);
        y = y + ROL(x<31:0>, 5) + t + Elem[w, e, 32];
        x<63:32> = ROL(x<63:32>, 30);
        <y, x> = ROL(y:x, 32);
    Q[d>>1] = x;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA1SU0

SHA1 schedule update 0.  
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	1	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

Encoding for the A1 variant

SHA1SU0.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	1	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

Encoding for the T1 variant

SHA1SU0.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    constant op1 = Q[d>>1]; op2 = Q[n>>1]; constant op3 = Q[m>>1];
    op2 = op2<63:0> : op1<127:64>;
    Q[d>>1] = op1 EOR op2 EOR op3;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# SHA1SU1

SHA1 schedule update 1.  
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	1	0	Vd			0	0	1	1	1	0	M	0	Vm					

### Encoding for the A1 variant

SHA1SU1.32 [<Qd>](#), [<Qm>](#)

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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	1	0		Vd				0	0	1	1	1	0	M	0		Vm		

### Encoding for the T1 variant

SHA1SU1.32 [<Qd>](#), [<Qm>](#)

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA1) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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();
    constant X = Q[d>>1]; constant Y = Q[m>>1];
    constant T = X EOR LSR(Y, 32);
    constant W0 = ROL(T<31:0>, 1);
    constant W1 = ROL(T<63:32>, 1);
    constant W2 = ROL(T<95:64>, 1);
    constant W3 = ROL(T<127:96>, 1) EOR ROL(T<31:0>, 2);
    Q[d>>1] = W3:W2:W1:W0;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# SHA256H

SHA256 hash update part 1.  
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	0	D	0	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

### Encoding for the A1 variant

SHA256H.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	0	D	0	0	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

### Encoding for the T1 variant

SHA256H.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    constant X = Q[d>>1]; constant Y = Q[n>>1]; constant W = Q[m>>1]; constant part1 = TRUE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256H2

SHA256 hash update part 2.  
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	0	D	0	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

Encoding for the A1 variant

SHA256H2.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	0	D	0	1	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

Encoding for the T1 variant

SHA256H2.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    constant X = Q[n>>1]; constant Y = Q[d>>1]; constant W = Q[m>>1]; constant part1 = FALSE;
    Q[d>>1] = SHA256hash(X, Y, W, part1);
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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					

### Encoding for the A1 variant

SHA256SU0.32 [<Qd>](#), [<Qm>](#)

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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					

### Encoding for the T1 variant

SHA256SU0.32 [<Qd>](#), [<Qm>](#)

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant m = 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
    bits(128) result;
    EncodingSpecificOperations(); CheckCryptoEnabled32();
    constant x = Q[d>>1]; constant y = Q[m>>1];
    constant t = y<31:0> : x<127:32>;
    for e = 0 to 3
        elt = Elem[t, e, 32];
        elt = ROR(elt, 7) EOR ROR(elt, 18) EOR LSR(elt, 3);
        Elem[result, e, 32] = elt + Elem[x, e, 32];
    Q[d>>1] = result;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SHA256SU1

SHA256 schedule update 1.  
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	0	D	1	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

Encoding for the A1 variant

SHA256SU1.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	0	D	1	0	Vn			Vd			1	1	0	0	N	Q	M	0	Vm					

Encoding for the T1 variant

SHA256SU1.32 [<Qd>](#), [<Qn>](#), [<Qm>](#)

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_SHA256) then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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();
    bits(32) elt;
    bits(128) result;
    constant x = Q[d>>1]; constant y = Q[n>>1]; constant z = Q[m>>1];
    constant T0 = z<31:0> : y<127:32>;

    T1 = z<127:64>;
    for e = 0 to 1
        elt = Elem[T1, e, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[x, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    T1 = result<63:0>;
    for e = 2 to 3
        elt = Elem[T1, e - 2, 32];
        elt = ROR(elt, 17) EOR ROR(elt, 19) EOR LSR(elt, 10);
        elt = elt + Elem[x, e, 32] + Elem[T0, e, 32];
        Elem[result, e, 32] = elt;

    Q[d>>1] = result;
```

## Operational information

If CPSR.DIT is 1:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = FALSE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = FALSE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant op1 = Elem[Din[n+r],e,esize];
      constant op2 = Elem[Din[m+r],e,esize];
      constant absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
      if long_destination then
        Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
      else
        Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	1	N	0	M	0	Vm				
size																															

### Encoding for the A1 variant

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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				0	1	0	1	N	0	M	0	Vm				
size																															

### Encoding for the T1 variant

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize];
            constant op2 = Elem[Din[m+r],e,esize];
            constant absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + absdiff;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + absdiff;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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”:
- | sz | <dt> |
|----|------|
| 0  | F32  |
| 1  | F16  |
- <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.
- <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();
    constant FPCR_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2, fpcr), fpcr);
```





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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (U == '1');  constant long_destination = FALSE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = FALSE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant op1 = Elem[Din[n+r],e,esize];
      constant op2 = Elem[Din[m+r],e,esize];
      constant absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
      if long_destination then
        Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
      else
        Elem[D[d+r],e,esize] = absdiff<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

31	30	29	28	27	26	25	24	23	22	21	20	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	1	N	0	M	0	Vm				
size																															

### Encoding for the A1 variant

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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				0	1	1	1	N	0	M	0	Vm				
size																															

### Encoding for the T1 variant

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize];
            constant op2 = Elem[Din[m+r],e,esize];
            constant absdiff = Abs(Int(op1,unsigned) - Int(op2,unsigned));
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = absdiff<2*esize-1:0>;
            else
                Elem[D[d+r],e,esize] = absdiff<esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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					

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant advsimd = TRUE;  constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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																															

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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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				

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant advsimd = TRUE; constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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					



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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

<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.

<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.

<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPAbs(Elem[D[m+r],e,esize], fpcr);
                else
                    constant result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
            else
                // VFP instruction
                constant FPCR\_Type fpcr = EffectiveFPCR();
                case esize of
                    when 16 H[d] = FPAbs(H[m], fpcr);
                    when 32 S[d] = FPAbs(S[m], fpcr);
                    when 64 D[d] = FPAbs(D[m], fpcr);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																															

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant or_equal = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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			
op																															

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant or_equal = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding T1: 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>&lt;dt&gt;</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						
<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.						
<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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = FPAbs(Elem[D[n+r],e,esize], fpcr);
            constant op2 = FPAbs(Elem[D[m+r],e,esize], fpcr);
            boolean test_passed;
            if or_equal then
                test_passed = FPCompareGE(op1, op2, fpcr);
            else
                test_passed = FPCompareGT(op1, op2, fpcr);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																															

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant or_equal = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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			
op																															

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant or_equal = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = FPAbs(Elem[D[n+r],e,esize], fpcr);
            constant op2 = FPAbs(Elem[D[m+r],e,esize], fpcr);
            boolean test_passed;
            if or_equal then
                test_passed = FPCompareGE(op1, op2, fpcr);
            else
                test_passed = FPCompareGT(op1, op2, fpcr);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																															

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			
op																															

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

- <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.
- <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.
- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

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

Operational information

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			
op																															

### 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			
op																															

### 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

- <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.
- <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.
- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

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

Operational information

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

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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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																															

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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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					

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:

sz	<dt>
0	F32
1	F16

<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.

<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.

<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], fpcr);
    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                H[d] = FPAdd(H[n], H[m], fpcr);
            when 32
                S[d] = FPAdd(S[n], S[m], fpcr);
            when 64
                D[d] = FPAdd(D[n], D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

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;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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
11	I64

- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] + Elem[D[m+r],e,esize];
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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

31	30	29	28	27	26	25	24	23	22	21	20	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 0 0			N	0	M	0	Vm							
size																															

Encoding for the A1 variant

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant elements = 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				0	1	0	0	N	0	M	0	Vm				
size																															

Encoding for the T1 variant

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant elements = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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();
    for e = 0 to elements-1
        constant result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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			0	N	0	M	0	Vm				
size												op																			

### Encoding for the A1 variant

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vaddw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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			0	N	0	M	0	Vm						
size												op																			

### Encoding for the T1 variant

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vaddw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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
        integer op1;
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        constant result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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 0			1	N	0	M	0	Vm						
size												op																			

#### Encoding for the A1 variant

VADDW{<c>}{<q>}.<dt> {<Qd>}, {<Qn>}, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vaddw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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 for the T1 variant

VADDW{<c>}{<q>}.<dt> {<Qd>}, {<Qn>}, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vaddw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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
        integer op1;
        if is_vaddw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        constant result = op1 + Int(Elem[Din[m],e,esize],unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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>

### 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																															

#### 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>

## T1

15	14	13	12	11	10	9	8	7	6	5	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																															

### 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>

## 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																															

### 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>

## Assembler Symbols

- <c> For encoding A1 and A2: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1 and T2: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <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](#).

## Operation

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

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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			

### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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					

### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.
- For encoding T1: see [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND D[m+r];

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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													

#### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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													

#### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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																															

### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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																															

### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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 encoding A1 and A2: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1 and T2: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.



- <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](#).

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
        D[d+r] = D[d+r] AND NOT(imm64);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

31	30	29	28	27	26	25	24	23	22	21	20	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	1	Vn				Vd				0 0 0 1				N	Q	M	1	Vm			

#### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	1	Vn			Vd			0	0	0	1	N	Q	M	1	Vm					

#### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
    for r = 0 to regs-1
        D[d+r] = D[n+r] AND NOT(D[m+r]);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																																		

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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
op																															

## 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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
for r = 0 to regs-1
    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]));
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	Vn				Vd				0				0	0	1	N	Q	M	1	Vm			
op																																		

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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			
op																															

## 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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
for r = 0 to regs-1
    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]));
```

## Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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			
op																																		

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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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			
op																															



## 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;
if op == '00' then SEE "VEOR";
if op == '01' then operation = VBitOps_VBSL;
if op == '10' then operation = VBitOps_VBIT;
if op == '11' then operation = VBitOps_VBIF;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
    for r = 0 to regs-1
        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]));
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 computation 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			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant integer esize = 16 << UInt(S);
constant elements = 64 DIV esize; constant regs = 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			

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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FCMA) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if S == '0' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant integer esize = 16 << UInt(S);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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
- <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.
- <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

## Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    constant operand1 = D[n+r];
    constant operand2 = D[m+r];
    constant FPCR_Type fpcr = StandardFPCR();
    for e = 0 to (elements DIV 2)-1
        bits(esize) element1;
        bits(esize) element3;
        case rot of
            when '0'
                element1 = FPNeg(Elem[operand2,e*2+1,esize], fpcr);
                element3 = Elem[operand2,e*2,esize];
            when '1'
                element1 = Elem[operand2,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,e*2,esize], fpcr);
        constant result1 = FPAdd(Elem[operand1,e*2,esize],element1,fpcr);
        constant result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,fpcr);
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *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	0	1	Vd			0	F	0	1	0	Q	M	0	Vm					

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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		

## 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer_size = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q>	See <i>Standard assembler syntax fields</i> .
-----	---

<dt>	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

<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.

**<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            boolean test_passed;
            if floating_point then
                constant bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPCR());
            else
                test_passed = (Elem[D[m+r],e,esize] == Zeros(esize));
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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	
1	1	1	1	0	0	1	1	0	D	size	Vn					Vd					1	0	0	0	N	Q	M	1	Vm			

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;
if size == '11' then UNDEFINED;
constant int_operation = TRUE; constant integer esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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			



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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant int_operation = FALSE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '11' then UNDEFINED;
constant int_operation = TRUE;  constant integer esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant int_operation = FALSE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For encoding A1 and T1: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	I8
01	I16
10	I32

For encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            boolean test_passed;
            if int_operation then
                test_passed = (op1 == op2);
            else
                test_passed = FPCompareEQ(op1, op2, StandardFPCR());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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		

### 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		

## 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer_size = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q>	See <i>Standard assembler syntax fields</i> .
-----	---

<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

<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.

<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            boolean test_passed;
            if floating_point then
                constant bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPCR());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) >= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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								

#### 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;
if size == '11' then UNDEFINED;
constant vtype = if U == '1' then VCGEType\_unsigned else VCGEType\_signed;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant vtype = VCGEType\_fp;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '11' then UNDEFINED;
constant vtype = if U == '1' then VCGEType\_unsigned else VCGEType\_signed;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	1	1	1	1	1	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			



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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant vtype = VCGEType fp;
constant integer esize = 32 >> UInt(sz);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For encoding A1 and T1: 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 encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            boolean test_passed;
            case vtype of
                when VCGEType\_signed    test_passed = (SInt(op1) >= SInt(op2));
                when VCGEType\_unsigned  test_passed = (UInt(op1) >= UInt(op2));
                when VCGEType\_fp        test_passed = FPCompareGE(op1, op2, StandardFPCR());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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			

#### 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		

## 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer_size = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see *Standard assembler syntax fields*.

<q>	See <i>Standard assembler syntax fields</i> .
-----	---

<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

<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.

<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            boolean test_passed;
            if floating_point then
                constant bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPCR());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) > 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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](#).

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						

#### 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;
if size == '11' then UNDEFINED;
constant vtype = if U == '1' then VCGTtype\_unsigned else VCGTtype\_signed;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	1	sz	Vn			Vd			1			1	1	0	N	Q	M	0	Vm			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant vtype = VCGTtype_fp;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '11' then UNDEFINED;
constant vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	1	1	1	1	1	0	D	1	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant vtype = VCGTtype_fp;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For encoding A1 and T1: 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 encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            boolean test_passed;
            case vtype of
                when VCGTtype\_signed    test_passed = (SInt(op1) > SInt(op2));
                when VCGTtype\_unsigned  test_passed = (UInt(op1) > UInt(op2));
                when VCGTtype\_fp        test_passed = FPCompareGT(op1, op2, StandardFPCR());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *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	0	1		Vd				0	F	0	1	1	Q	M	0		Vm		

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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		

## 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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: 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

**<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.

<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            boolean test_passed;
            if floating_point then
                constant bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPCR());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) <= 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

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>

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

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>
```

T1

15	14	13	12	11	10	9	8	7	6	5	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				

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>
```

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	0	D	0	sz	Vn				Vd				1	1	1	0	N	Q	M	0	Vm			

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

- <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.
- <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.
- <c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For encoding A1 and T1: 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 encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

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

Operational information

The description of *VCGE\_r* gives the operational pseudocode for this instruction.

VCLS

Vector Count Leading Sign Bits counts 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 count 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	0	0	Vd			0	1	0	0	0	Q	M	0	Vm					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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		

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```



Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingSignBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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 *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	0	1		Vd				0	F	1	0	0	Q	M	0		Vm		

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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		

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            boolean test_passed;
            if floating_point then
                constant bits(esize) zero = FPZero('0', esize);
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPCR());
            else
                test_passed = (SInt(Elem[D[m+r],e,esize]) < 0);
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						

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		

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				

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	0	Vm					

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

- <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.
- <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.
- <c> For encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For encoding A1 and T1: 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 encoding A2 and T2: is the data type for the elements of the vectors, encoded in “sz”:

sz	<dt>
0	F32
1	F16

- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.

Operation

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

Operational information

The description of *VCGT\_r* gives the operational pseudocode for this instruction.

# 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 *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	0	1	Q	M	0	Vm					

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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	1	Q	M	0		Vm		

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```



Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	I8
01	I16
10	I32
11	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.
- <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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = CountLeadingZeroBits(Elem[D[m+r],e,esize])<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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 computation 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			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant integer esize = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then UNDEFINED;
constant elements = 64 DIV esize;
constant regs = 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			

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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FCMA) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant integer esize = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then UNDEFINED;
constant elements = 64 DIV esize;
constant regs = 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
- <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.
- <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

## Operation

```
EncodingSpecificOperations();
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
    constant operand1 = D[n+r];
    constant operand2 = D[m+r];
    constant operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        bits(esize) element1;
        bits(esize) element2;
        bits(esize) element3;
        bits(esize) element4;
        case rot of
            when '00'
                element1 = Elem[operand2,e*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,e*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,e*2+1,esize], fpcr);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,e*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,e*2,esize], fpcr);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,e*2+1,esize], fpcr);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,e*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,e*2,esize], fpcr);
                element4 = Elem[operand1,e*2+1,esize];
        constant result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, fpcr);
        constant result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3, fpcr);
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 computation 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		

#### 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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant m = if S=='1' then UInt(M:Vm) else UInt(Vm);
constant integer esize = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then UNDEFINED;
constant elements = 64 DIV esize;
constant regs = if Q == '0' then 1 else 2;
constant index = if S=='1' then 0 else UInt(M);
```

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				

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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FCMA) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant m = if S=='1' then UInt(M:Vm) else UInt(Vm);
constant integer esize = 16 << UInt(S);
if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then UNDEFINED;
constant elements = 64 DIV esize;
constant regs = if Q == '0' then 1 else 2;
constant index = if S=='1' then 0 else UInt(M);
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <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.
- <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 half-precision scalar variant: is the 64-bit name of the second SIMD&FP source register, encoded in the "Vm" field.  
For the single-precision scalar variant: 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".

rot	<rotate>
00	0
01	90
10	180
11	270

## Operation

```

EncodingSpecificOperations();
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
    constant operand1 = D[n+r];
    constant operand2 = Din[m];
    constant operand3 = D[d+r];
    for e = 0 to (elements DIV 2)-1
        bits(esize) element1;
        bits(esize) element2;
        bits(esize) element3;
        bits(esize) element4;
        case rot of
            when '00'
                element1 = Elem[operand2,index*2,esize];
                element2 = Elem[operand1,e*2,esize];
                element3 = Elem[operand2,index*2+1,esize];
                element4 = Elem[operand1,e*2,esize];
            when '01'
                element1 = FPNeg(Elem[operand2,index*2+1,esize], fpcr);
                element2 = Elem[operand1,e*2+1,esize];
                element3 = Elem[operand2,index*2,esize];
                element4 = Elem[operand1,e*2+1,esize];
            when '10'
                element1 = FPNeg(Elem[operand2,index*2,esize], fpcr);
                element2 = Elem[operand1,e*2,esize];
                element3 = FPNeg(Elem[operand2,index*2+1,esize], fpcr);
                element4 = Elem[operand1,e*2,esize];
            when '11'
                element1 = Elem[operand2,index*2+1,esize];
                element2 = Elem[operand1,e*2+1,esize];
                element3 = FPNeg(Elem[operand2,index*2,esize], fpcr);
                element4 = Elem[operand1,e*2+1,esize];
        constant result1 = FPMulAdd(Elem[operand3,e*2,esize],element2,element1, fpcr);
        constant result2 = FPMulAdd(Elem[operand3,e*2+1,esize],element4,element3,fpcr);
        Elem[D[d+r],e*2,esize] = result1;
        Elem[D[d+r],e*2+1,esize] = result2;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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												E																			



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

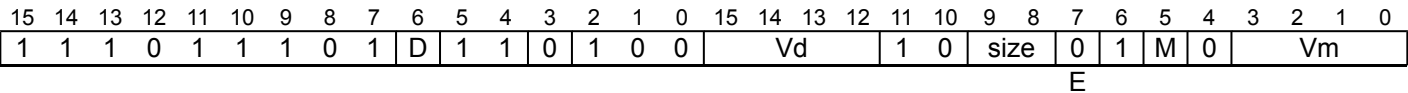
```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = integer UNKNOWN;
```

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



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

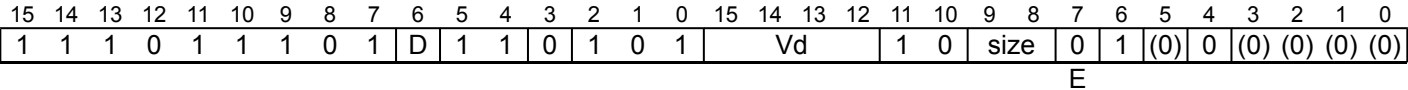
```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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



**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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = integer UNKNOWN;
```

**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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    bits(4) nzcvc;
    case esize of
        when 16
            constant bits(16) op16 = if with_zero then FPZero('0', 16) else H[m];
            nzcvc = FPCompare(H[d], op16, quiet_nan_exc, fpcr);
        when 32
            constant bits(32) op32 = if with_zero then FPZero('0', 32) else S[m];
            nzcvc = FPCompare(S[d], op32, quiet_nan_exc, fpcr);
        when 64
            constant bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
            nzcvc = FPCompare(D[d], op64, quiet_nan_exc, fpcr);

    FPSCR<31:28> = nzcvc; // FPSCR.<N,Z,C,V> set to nzcvc
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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												E																			

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)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

Decode for all variants of this encoding

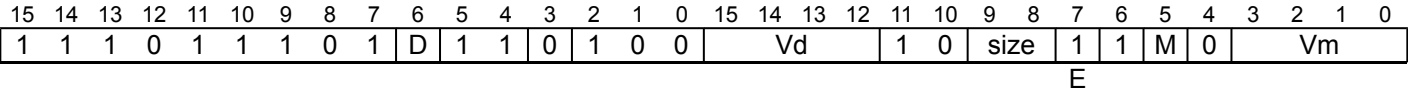
```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = integer UNKNOWN;
```

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



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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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)	(0)	
																								E							

## 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)

```
VCMPE{<c>}{<q>}.F64 <Dd>, #0.0
```

## Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant quiet_nan_exc = (E == '1'); constant with_zero = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = integer UNKNOWN;
```

## 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    bits(4) nzcvc;
    case esize of
        when 16
            constant bits(16) op16 = if with_zero then FPZero('0', 16) else H[m];
            nzcvc = FPCompare(H[d], op16, quiet_nan_exc, fpcr);
        when 32
            constant bits(32) op32 = if with_zero then FPZero('0', 32) else S[m];
            nzcvc = FPCompare(S[d], op32, quiet_nan_exc, fpcr);
        when 64
            constant bits(64) op64 = if with_zero then FPZero('0', 64) else D[m];
            nzcvc = FPCompare(D[d], op64, quiet_nan_exc, fpcr);

    FPSCR<31:28> = nzcvc; // FPSCR.<N,Z,C,V> set to nzcvc
```

## 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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		

#### Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

#### Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

#### Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8;  constant elements = 8;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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		

#### Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

#### Encoding for the 128-bit SIMD vector variant

Applies when (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

#### Decode for all variants of this encoding

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8;  constant elements = 8;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = BitCount(Elem[D[m+r],e,esize])<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																size															

#### 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

```
constant double_to_single = (size == '11');
constant d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
constant m = 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			
																size															

#### 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

```
constant double_to_single = (size == '11');
constant d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);
constant m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

## 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.
<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.
<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);
    if double_to_single then
        S[d] = FPConvert(D[m], EffectiveFPCR(), 32);
    else
        D[d] = FPConvert(S[m], EffectiveFPCR(), 64);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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				1	1	op	0	Q	M	1	Vm				

#### Encoding for the 64-bit SIMD vector variant

Applies when (imm6 != 000xxx && Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>

#### Encoding for the 128-bit SIMD vector variant

Applies when (imm6 != 000xxx && Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

#### Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant to_fixed = (op<0> == '1'); constant frac_bits = 64 - UInt(imm6);
constant unsigned = (U == '1');
constant integer esize = 16 << UInt(op<1>);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				1	1	op	0	Q	M	1	Vm				

Encoding for the 64-bit SIMD vector variant

Applies when (imm6 != 000xxx && Q == 0)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>
```

Encoding for the 128-bit SIMD vector variant

Applies when (imm6 != 000xxx && Q == 1)

```
VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>
```

Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if op<1> == '0' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if op<1> == '0' && imm6 == '10xxxx' then UNDEFINED;
if imm6 == '0xxxxx' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant to_fixed = (op<0> == '1'); constant frac_bits = 64 - UInt(imm6);
constant unsigned = (U == '1');
constant integer esize = 16 << UInt(op<1>);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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)*.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(esize) result;
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[m+r],e,esize];
            if to_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, fpcr,
                                   FPRounding\_ZERO, esize);
            else
                result = FixedToFP(op1, frac_bits, unsigned, fpcr,
                                   FPRounding\_TIEEVEN, esize);
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



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																															

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;
if sf == '01' && cond != '1110' then UNPREDICTABLE;
constant to_fixed = (op == '1'); constant unsigned = (U == '1');
constant integer size = if sx == '0' then 16 else 32;
constant frac_bits = size - UInt(imm4:i);
constant integer fp_size = 8 << UInt(sf);
constant integer d = if sf == '11' then UInt(D:Vd) else UInt(Vd:D);
if frac_bits < 0 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	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	op	1	U	Vd			1	0	sf		sx	1	i	0	imm4				

### 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;
if sf == '01' && InITBlock() then UNPREDICTABLE;
constant to_fixed = (op == '1'); constant unsigned = (U == '1');
constant integer size = if sx == '0' then 16 else 32;
constant frac_bits = size - UInt(imm4:i);
constant integer fp_size = 8 << UInt(sf);
constant integer d = if sf == '11' then UInt(D:Vd) else UInt(Vd:D);

if frac_bits < 0 then UNPREDICTABLE;
```

### 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.
- <Ddm> Is the 64-bit name of the SIMD&FP destination and source register, encoded in the "D:Vd" 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].

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();    CheckVFPEnabled(TRUE);
    constant FPCR_Type fpcr = EffectiveFPCR();
    if to_fixed then
        bits(size) result;
        case fp_size of
            when 16
                result = FPToFixed(H[d], frac_bits, unsigned, fpcr, FPRounding_ZERO, size);
                S[d] = Extend(result, 32, unsigned);
            when 32
                result = FPToFixed(S[d], frac_bits, unsigned, fpcr, FPRounding_ZERO, size);
                S[d] = Extend(result, 32, unsigned);
            when 64
                result = FPToFixed(D[d], frac_bits, unsigned, fpcr, FPRounding_ZERO, size);
                D[d] = Extend(result, 64, unsigned);
        else
            case fp_size of
                when 16
                    H[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, fpcr, FPRounding_TIEEVEN, 16);
                when 32
                    S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, fpcr, FPRounding_TIEEVEN, 32);
                when 64
                    D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, fpcr, FPRounding_TIEEVEN, 64);
```

## 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				

#### 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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant to_integer = (op<1> == '1'); constant unsigned = (op<0> == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		

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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant to_integer = (op<1> == '1'); constant unsigned = (op<0> == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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(); CheckAdvSIMDEnabled();
    constant FPCR\_Type fpcr = StandardFPCR();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, fpcr, FPRounding\_ZERO, esize);
            else
                result = FixedToFP(op1, 0, unsigned, fpcr, FPRounding\_TIEEVEN, esize);
            Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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			1	1	op	0	0	M	0	Vm			

#### Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)
```

#### Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)
```

#### Decode for all variants of this encoding

```
if size != '01' then UNDEFINED;
constant half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
constant esize = 16; constant elements = 4;
constant m = UInt(M:Vm); constant d = 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		

#### Encoding for the Half-precision to single-precision variant

Applies when (op == 1)

```
VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)
```

#### Encoding for the Single-precision to half-precision variant

Applies when (op == 0)

```
VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)
```

#### Decode for all variants of this encoding

```
if size != '01' then UNDEFINED;
constant half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
constant esize = 16; constant elements = 4;
constant m = UInt(M:Vm); constant d = UInt(D:Vd);
```



## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for e = 0 to elements-1
        if half_to_single then
            Elem[Q[d>>1],e,32] = FPConvert(Elem[Din[m],e,16], fpcr, 32);
        else
            Elem[D[d],e,16] = FPConvert(Elem[Qin[m>>1],e,32], fpcr, 16);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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										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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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				
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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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(); CheckVFPEEnabled(TRUE);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    constant FPRounding rounding = if zero_rounding then FPRounding\_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
    else
        case esize of
            when 16
                H[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 16);
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 64);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	1	0	0	1	M	0	Vm				

#### Encoding for the A1 variant

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

#### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer m = 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				

#### Encoding for the T1 variant

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

#### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer m = UInt(M:Vm);
```

### Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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.
- <Qm> Is the 128-bit name of the SIMD&FP source register, encoded in the "M:Vm" field as <Qm>\*2.

## Operation

```
bits(128) operand;
bits(64) result;

if ConditionPassed\(\) then
    EncodingSpecificOperations\(\);
    CheckAdvSIMDEnabled\(\);
    constant FPCR\_Type fpcr = StandardFPCR\(\);

    operand = Q[m>>1];
    for e = 0 to 3
        constant bits(32) op = Elem[operand, e, 32];
        Elem[result, e, 16] = FPConvertBF(op, fpcr);
    D[d] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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](#), [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	0	0	0	Vd				1	0	size	op	1	M	0	Vm				
cond												opc2																			

#### Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

#### Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

#### 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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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			
opc2																															



Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

```
VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>
```

Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>
```

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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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.

<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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    constant FPRounding rounding = if zero_rounding then FPRounding\_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
    else
        case esize of
            when 16
                H[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 16);
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 64);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
																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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		
																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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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

<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.

<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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize],
                                     0, unsigned, fpcr, rounding, esize);
```



## 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	!= 00	op	1	M	0	Vm					
RM															size																

#### 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' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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	!= 00	op	1	M	0	Vm					
RM															size																

#### 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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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

- <q> See *Standard assembler syntax fields*.
- <dt> Is the data type for the elements of the destination, 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.
- <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
```

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																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

```
constant uses_double = (sz == '1'); constant convert_from_half = (op == '0');
constant integer lowbit = (if T == '1' then 16 else 0);
constant integer d = if uses_double && convert_from_half then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if uses_double && !convert_from_half 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	0	1	op	Vd				1	0	1	sz	0	1	M	0	Vm			
																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

```
constant uses_double = (sz == '1'); constant convert_from_half = (op == '0');
constant integer lowbit = (if T == '1' then 16 else 0);
constant integer d = if uses_double && convert_from_half then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if uses_double && !convert_from_half then UInt(M:Vm) else 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.
- <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.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEntered(TRUE);
    bits(16) hp;
    constant FPCR_Type fpcr = EffectiveFPCR();
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, fpcr, 64);
        else
            S[d] = FPConvert(hp, fpcr, 32);
    else
        if uses_double then
            hp = FPConvert(D[m], fpcr, 16);
        else
            hp = FPConvert(S[m], fpcr, 16);
        S[d]<lowbit+15:lowbit> = hp;
```



## 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)

31	30	29	28	27	26	25	24	23	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	1	Vd				1	0	0	1	0	1	M	0	Vm			
cond																															

#### Encoding for the A1 variant

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

#### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
constant integer d = UInt(Vd:D);
constant integer m = UInt(Vm:M);
```

### 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	0	1	1	1	0	1	D	1	1	0	0	1	1	Vd				1	0	0	1	0	1	M	0	Vm			

#### Encoding for the T1 variant

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

#### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
constant integer d = UInt(Vd:D);
constant integer m = 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());
```



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				
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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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			
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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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

<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.

<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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize],
                                     0, unsigned, fpcr, rounding, esize);
```



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	!= 00	op	1	M	0	Vm			
RM																size															

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' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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	!= 00	op	1	M	0	Vm					
RM															size																

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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, 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.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
```

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			
																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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		
																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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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

<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.

<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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize],
                                     0, unsigned, fpcr, rounding, esize);
```



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	!= 00	op	1	M	0	Vm				
RM																size															

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' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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	!= 00	op	1	M	0	Vm				
RM																size															

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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, 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.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
```

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		
																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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		
																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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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

<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.

<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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    Elem[D[d+r],e,esize] = FPToFixed(Elem[D[m+r],e,esize],
                                     0, unsigned, fpcr, rounding, esize);
```





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	!= 00	op	1	M	0	Vm				
RM																size															

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' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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	!= 00	op	1	M	0	Vm				
RM															size																

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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant unsigned = (op == '0');
constant d = UInt(Vd:D);
constant integer esize = 8 << UInt(size);
constant integer m = 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

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, 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.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
```

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*, *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		0	1	M	0	Vm			
cond										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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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				
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";
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant to_integer = (opc2<2> == '1');
constant integer m = if to_integer && size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer d = if !to_integer && size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant boolean unsigned = (if to_integer then opc2<0> else op) == '0';
constant boolean zero_rounding = 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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(); CheckVFPEEnabled(TRUE);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    constant FPRounding rounding = if zero_rounding then FPRounding\_ZERO else FPRoundingMode(fpcr);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(H[m], 0, unsigned, fpcr, rounding, 32);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, fpcr, rounding, 32);
    else
        case esize of
            when 16
                H[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 16);
            when 32
                S[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 32);
            when 64
                D[d] = FixedToFP(S[m], 0, unsigned, fpcr, rounding, 64);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																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

```
constant uses_double = (sz == '1'); constant convert_from_half = (op == '0');  
constant integer lowbit = (if T == '1' then 16 else 0);  
constant integer d = if uses_double && convert_from_half then UInt(D:Vd) else UInt(Vd:D);  
constant integer m = if uses_double && !convert_from_half 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	0	1	op	Vd				1	0	1	sz	1	1	M	0	Vm			
																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

```
constant uses_double = (sz == '1'); constant convert_from_half = (op == '0');
constant integer lowbit = (if T == '1' then 16 else 0);
constant integer d = if uses_double && convert_from_half then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if uses_double && !convert_from_half then UInt(M:Vm) else 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.
- <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.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEntered(TRUE);
    bits(16) hp;
    constant FPCR_Type fpcr = EffectiveFPCR();
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, fpcr, 64);
        else
            S[d] = FPConvert(hp, fpcr, 32);
    else
        if uses_double then
            hp = FPConvert(D[m], fpcr, 16);
        else
            hp = FPConvert(S[m], fpcr, 16);
        S[d]<lowbit+15:lowbit> = hp;
```



# 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)

31	30	29	28	27	26	25	24	23	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	1	Vd				1	0	0	1	1	1	M	0	Vm			
cond																															

### Encoding for the A1 variant

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
constant integer d = UInt(Vd:D);
constant integer m = UInt(Vm:M);
```

## 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	0	1	1	1	0	1	D	1	1	0	0	1	1	Vd				1	0	0	1	1	1	M	0	Vm			

### Encoding for the T1 variant

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
constant integer d = UInt(Vd:D);
constant integer m = 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());
```



## 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																															

#### 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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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				

#### 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 UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            H[d] = FPDIV(H[n], H[m], fpcr);
        when 32
            S[d] = FPDIV(S[n], S[m], fpcr);
        when 64
            D[d] = FPDIV(D[n], D[m], fpcr);
```

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					

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = 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					

## 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = if Q == '1' then 2 else 1;
```

## 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.
<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.

## Operation

```
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();

bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        constant bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        constant bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        constant bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
        constant bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
        constant bits(32) sum = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
                                             BFMulH(elt1_b, elt2_b, fpcr), fpcr);
        Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum, fpcr);
D[d+r] = result;
```



VDOT (vector)

BF16 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)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	N	Q	M	0	Vm			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = 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	0	0	0	D	0	0	Vn			Vd			1	1	0	1	N	Q	M	0	Vm					

## 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = if Q == '1' then 2 else 1;
```

## Assembler Symbols

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

```
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();

bits(64) operand1;
bits(64) operand2;
bits(64) result;

for r = 0 to regs-1
    operand1 = Din[n+r];
    operand2 = Din[m+r];
    result = Din[d+r];
    for e = 0 to 1
        constant bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
        constant bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
        constant bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
        constant bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
        constant bits(32) sum = FPAdd_BF16(BFMulH(elt1_a, elt2_a, fpcr),
                                             BFMulH(elt1_b, elt2_b, fpcr), fpcr);
        Elem[result, e, 32] = FPAdd_BF16(Elem[result, e, 32], sum, fpcr);
    D[d+r] = result;
```

# 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																															

### Encoding for the A1 variant

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

### Decode for this encoding

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant t = UInt(Rt); constant regs = if Q == '0' then 1 else 2;
if B:E == '11' then UNDEFINED;
constant integer esize = 32 >> UInt(B:E);
constant integer elements = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

## T1

15	14	13	12	11	10	9	8	7	6	5	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)

### Encoding for the T1 variant

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)

VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

### Decode for this encoding

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant d = UInt(D:Vd); constant t = UInt(Rt); constant regs = if Q == '0' then 1 else 2;
if B:E == '11' then UNDEFINED;
constant integer esize = 32 >> UInt(B:E);
constant integer elements = 64 DIV esize;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<size>	The data size for the elements of the destination vector. It must be one of: <b>8</b> Encoded as [b, e] = 0b10.  <b>16</b> Encoded as [b, e] = 0b01.  <b>32</b> Encoded as [b, e] = 0b00.
<Qd>	The destination vector for a quadword operation.
<Dd>	The destination vector for a doubleword operation.
<Rt>	The Arm source register.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant scalar = R[t]<esize-1:0>;
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

### 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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant integer lsb = LowestSetBit(imm4<2:0>);
constant integer esize = 8 << lsb;
constant integer index = UInt(imm4<3:lsb+1>);
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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			

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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant integer lsb = LowestSetBit(imm4<2:0>);
constant integer esize = 8 << lsb;
constant integer index = UInt(imm4<3:lsb+1>);
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <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>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant scalar = Elem[D[m],index,esize];
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = scalar;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



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			

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;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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			

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;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.



For encoding T1: see [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
    for r = 0 to regs-1
        D[d+r] = D[n+r] EOR D[m+r];
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

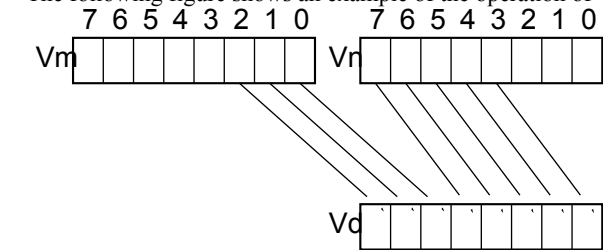
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 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;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
constant quadword_operation = (Q == '1');
constant integer_position = 8 * UInt(imm4);
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = 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;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
constant quadword_operation = (Q == '1');
constant integer_position = 8 * UInt(imm4);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
```

## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
<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 64-bit SIMD vector variant: 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 128-bit SIMD vector variant: 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.

## 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>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<size>	For the 64-bit SIMD vector variant: is the size of the operation, and can be one of 16 or 32. For the 128-bit SIMD vector variant: is the size of the operation, and can be one of 16, 32 or 64.
<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.
<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 64-bit SIMD vector variant: 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 128-bit SIMD vector variant: 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.

Operation

The description of [VEXT \(byte elements\)](#) gives the operational pseudocode for this instruction.

## 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](#), [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	0	N	Q	M	1	Vm			
op																															

#### 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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE; constant opl_neg = (op == '1');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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												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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE; constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

## 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					
op																															

## 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

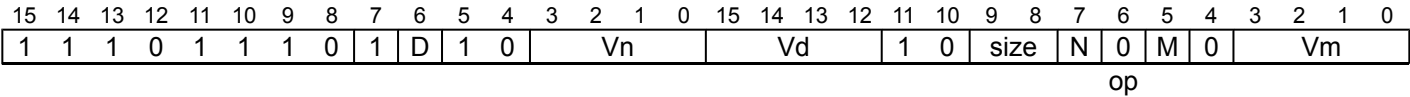
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE; constant opl_neg = (op == '1');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant advsimd = FALSE; constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.
<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
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(esize) op1 = Elem[D[n+r],e,esize];
            if op1_neg then op1 = FPNeg(op1, fpcr);
            Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                         op1, Elem[D[m+r],e,esize], fpcr);

else // VFP instruction
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant op16 = if op1_neg then FPNeg(H[n], fpcr) else H[n];
            H[d] = FPMulAdd(H[d], op16, H[m], fpcr);
        when 32
            constant op32 = if op1_neg then FPNeg(S[n], fpcr) else S[n];
            S[d] = FPMulAdd(S[d], op32, S[m], fpcr);
        when 64
            constant op64 = if op1_neg then FPNeg(D[n], fpcr) else D[n];
            D[d] = FPMulAdd(D[d], op64, D[m], fpcr);

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAB, VFMAT (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)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					

Encoding for the A1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<2:0>);
constant integer i = UInt(M:Vm<3>);
constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

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	1	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					

Encoding for the T1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<2:0>);
constant integer i = UInt(M:Vm<3>);
constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

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();
constant FPCR_Type fpcr = StandardFPCR();

constant bits(128) operand1 = Q[n>>1];
constant bits(64) operand2 = D[m];
constant bits(128) operand3 = Q[d>>1];
bits(128) result;

constant bits(32) element2 = Elem[operand2, i, 16] : Zeros(16);

for e = 0 to elements-1
    constant bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    constant bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2, fpcr);

Q[d>>1] = result;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMAB, VFMAT (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)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					

Encoding for the A1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

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	1	1	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					

Encoding for the T1 variant

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer elements = 128 DIV 32;
constant integer sel = UInt(Q);
```

Assembler Symbols

<bt> Is the bottom or top element specifier, encoded in “Q”:

Q	<bt>
0	B
1	T

<q> See *Standard assembler syntax fields*.

- <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();
constant FPCR_Type fpcr = StandardFPCR();

constant bits(128) operand1 = Q[n>>1];
constant bits(128) operand2 = Q[m>>1];
constant bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
    constant bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    constant bits(32) element2 = Elem[operand2, 2 * e + sel, 16] : Zeros(16);
    constant bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2, fpcr);

Q[d>>1] = result;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
S																															

#### 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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

constant integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = 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			
S																															

#### 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

constant integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = if Q == '0' then 1 else 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.
<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.
<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 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

## Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
constant FPCR_Type fpcr = 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>;
element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1, fpcr);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, fpcr);
    D[d+r] = result;
```





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)

Table with 32 columns (31 to 0) and 1 row. Bit 31 is 1, bit 30 is 1, bit 29 is 1, bit 28 is 1, bit 27 is 1, bit 26 is 1, bit 25 is 0, bit 24 is 0, bit 23 is 0, bit 22 is D, bit 21 is 1, bit 20 is 0, bits 18-16 are Vn, bits 15-12 are Vd, bit 11 is 1, bit 10 is 0, bit 9 is 0, bit 8 is 0, bit 7 is N, bit 6 is Q, bit 5 is M, bit 4 is 1, bits 3-0 are Vm.

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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = if Q == '0' then 1 else 2;
```

T1
(FEAT\_FHM)

Table with 32 columns (15 to 0 and 15 to 0) and 1 row. Bit 15 is 1, bit 14 is 1, bit 13 is 1, bit 12 is 1, bit 11 is 1, bit 10 is 1, bit 9 is 0, bit 8 is 0, bit 7 is 0, bit 6 is D, bit 5 is 1, bit 4 is 0, bits 3-0 are Vn, bits 15-12 are Vd, bit 11 is 1, bit 10 is 0, bit 9 is 0, bit 8 is 0, bit 7 is N, bit 6 is Q, bit 5 is M, bit 4 is 1, bits 3-0 are Vm.

## 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
constant integer esize = 32;
constant integer regs = 1 << UInt(Q);
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
```

## 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.
<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.
<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.

## Operation

```
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();

bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

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>;

for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1, fpcr);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, fpcr);
    D[d+r] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

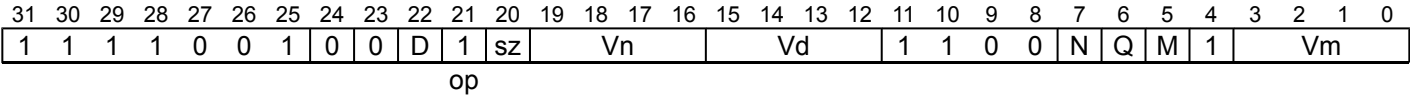
VFMS

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



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

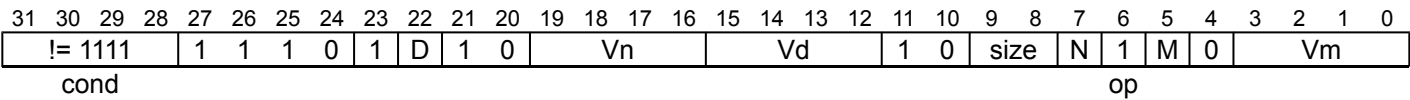
Applies when (Q == 1)

```
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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE; constant opl_neg = (op == '1');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

A2



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

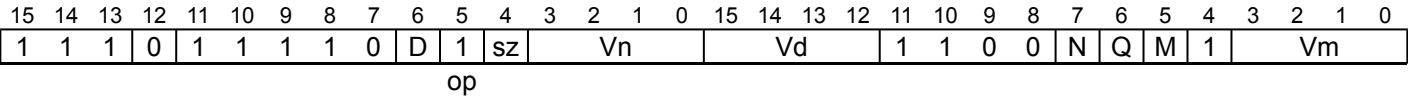
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE; constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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



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

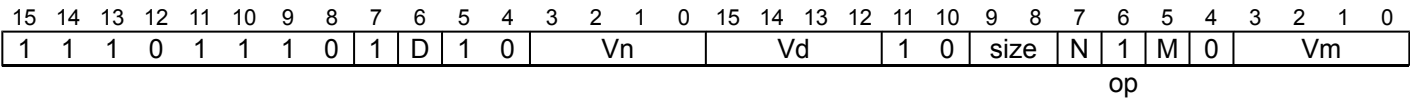
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE; constant opl_neg = (op == '1');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant advsimd = FALSE; constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:
- | sz | <dt> |
|----|------|
| 0  | F32  |
| 1  | F16  |
- <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.
- <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.
- <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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                bits(esize) op1 = Elem[D[n+r],e,esize];
                if op1_neg then op1 = FPNeg(op1, fpcr);
                Elem[D[d+r],e,esize] = FPMulAdd(Elem[D[d+r],e,esize],
                                                op1, Elem[D[m+r],e,esize], fpcr);

    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                constant op16 = if op1_neg then FPNeg(H[n], fpcr) else H[n];
                H[d] = FPMulAdd(H[d], op16, H[m], fpcr);
            when 32
                constant op32 = if op1_neg then FPNeg(S[n], fpcr) else S[n];
                S[d] = FPMulAdd(S[d], op32, S[m], fpcr);
            when 64
                constant op64 = if op1_neg then FPNeg(D[n], fpcr) else D[n];
                D[d] = FPMulAdd(D[d], op64, D[m], fpcr);

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
S																															

### 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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

constant integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = 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			
S																															

### 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

constant integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = if Q == '0' then 1 else 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.
<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.
<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 64-bit SIMD vector variant: is the element index in the range 0 to 1, encoded in the "Vm<3>" field. For the 128-bit SIMD vector variant: is the element index in the range 0 to 3, encoded in the "M:Vm<3>" field.

## Operation

```
CheckAdvSIMDEnabled();
bits(datasize) operand1 ;
bits(datasize) operand2 ;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;
constant FPCR_Type fpcr = 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>;
element2 = Elem[operand2, index, esize DIV 2];
for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1, fpcr);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, fpcr);
    D[d+r] = result;
```



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																															

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;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
constant integer esize = 32;
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
constant integer regs = 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																															

## 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

constant integer d = UInt(D:Vd);
constant integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
constant integer esize = 32;
constant integer regs = 1 << UInt(Q);
constant integer datasize = 32 << UInt(Q);
constant boolean sub_op = S == '1';
```

## 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.
<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.
<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.

## Operation

```
CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();

bits(datasize) operand1;
bits(datasize) operand2;
bits(64) operand3;
bits(64) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2;

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>;

for r = 0 to regs-1
    operand3 = D[d+r];
    for e = 0 to 1
        element1 = Elem[operand1, 2*r+e, esize DIV 2];
        element2 = Elem[operand2, 2*r+e, esize DIV 2];
        if sub_op then element1 = FPNeg(element1, fpcr);
        Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], element1, element2, fpcr);
    D[d+r] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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												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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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				
																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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant op16 = if op1_neg then FPNeg(H[n], fpcr) else H[n];
            H[d] = FPMulAdd(FPNeg(H[d], fpcr), op16, H[m], fpcr);
        when 32
            constant op32 = if op1_neg then FPNeg(S[n], fpcr) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d], fpcr), op32, S[m], fpcr);
        when 64
            constant op64 = if op1_neg then FPNeg(D[n], fpcr) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d], fpcr), op64, D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## VFNMS

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												op																			

#### Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

#### Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

#### Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

#### Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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	0	M	0	Vm				
																op															

## Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

```
VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

## Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

## Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

## Decode for all variants of this encoding

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant opl_neg = (op == '1');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant op16 = if op1_neg then FPNeg(H[n], fpcr) else H[n];
            H[d] = FPMulAdd(FPNeg(H[d], fpcr), op16, H[m], fpcr);
        when 32
            constant op32 = if op1_neg then FPNeg(S[n], fpcr) else S[n];
            S[d] = FPMulAdd(FPNeg(S[d], fpcr), op32, S[m], fpcr);
        when 64
            constant op64 = if op1_neg then FPNeg(D[n], fpcr) else D[n];
            D[d] = FPMulAdd(FPNeg(D[d], fpcr), op64, D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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	0	N	Q	M	0	Vm			
op																																

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;
if size == '11' then UNDEFINED;
constant add = (op == '0');  constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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																															

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;
if size == '11' then UNDEFINED;
constant add = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Int(Elem[D[n+r],e,esize], unsigned);
            constant op2 = Int(Elem[D[m+r],e,esize], unsigned);
            constant result = (if add then op1+op2 else op1-op2) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 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	0	Vm				
op																															

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;
if size == '11' then UNDEFINED;
constant add = (op == '0');  constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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																															



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;
if size == '11' then UNDEFINED;
constant add = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Int(Elem[D[n+r],e,esize], unsigned);
            constant op2 = Int(Elem[D[m+r],e,esize], unsigned);
            constant result = (if add then op1+op2 else op1-op2) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	0	1	1	M	0	Vm			

Encoding for the A1 variant

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant d = UInt(Vd:D); constant m = UInt(Vm:M);
```

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	1	1	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	0	1	1	M	0	Vm			

Encoding for the T1 variant

VINS{<q>}.F16 <Sd>, <Sm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant d = UInt(Vd:D); constant m = 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);  
    S[d]<31:16> = H[m];
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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  $2^{32}$ , 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
!= 1111				1	1	1	0	1	D	1	1	1	0	0	1	Vd				1	0	1	1	1	1	M	0	Vm			
cond																															

Encoding for the A1 variant

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
constant d = UInt(Vd:D);  constant m = 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			

Encoding for the T1 variant

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_JSCVT) then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
constant d = UInt(Vd:D);  constant m = 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();  
CheckVFPEnabled(TRUE);  
constant bits(64) fltval = D[m];  
bits(32) intval;  
bit      Z;  
(intval, Z) = FPToFixedJS(fltval, EffectiveFPCR(), 32);  
FPSCR<31:28> = '0':Z:'00';  
S[d] = intval;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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							

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

```
constant regs = 1; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size);
constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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							

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

```
constant regs = 2; if align == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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					



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

```
constant regs = 3; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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					

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

```
constant regs = 4;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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				

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

```
constant regs = 1; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size);
constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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					

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

```
constant regs = 2; if align == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	0	size		align		Rm					

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

```
constant regs = 3; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	0	size		align		Rm			

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

```
constant regs = 4;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant integer ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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 encoding A1, A2, A3 and A4: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2, T3 and T4: 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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for r = 0 to regs-1
        for e = 0 to elements-1
            bits(ebytes*8) data;
            if ebytes != 8 then
                data = MemU[address, ebytes];
            else
                if !IsAligned(address, ebytes) && AlignmentEnforced() then
                    constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
                    AArch32.Abort(fault);

                if BigEndian(AccessType_ASIMD) then
                    data<31:0> = MemU[address+4, 4];
                    data<63:32> = MemU[address, 4];
                else
                    data<31:0> = MemU[address, 4];
                    data<63:32> = MemU[address+4, 4];

                Elem[D[d+r], e, 8*ebytes] = data;
                address = address + ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 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				

#### 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;
constant ebytes = 1 << UInt(size); constant regs = if T == '0' then 1 else 2;
constant alignment = if a == '0' then 1 else ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

### 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				



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;
constant ebytes = 1 << UInt(size);  constant regs = if T == '0' then 1 else 2;
constant alignment = if a == '0' then 1 else ebytes;
constant d = UInt(D:Vd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	RESERVED

<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();

  constant address = R[n];

  constant boolean nontemporal = FALSE;
  constant boolean privileged = PSTATE.EL != EL0;
  constant boolean tagchecked = FALSE;
  constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
    tagchecked, privileged);

  if !IsAligned(address, alignment) then
    constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
    AArch32.Abort(fault);

  constant integer esize = 8 * ebytes;
  constant bits(esize) element = MemU[address,ebytes];
  constant bits(64) replicated_element = Replicate(element, 64 DIV esize);
  for r = 0 to regs-1
    D[d+r] = replicated_element;
  if wback then
    if register_index then
      R[n] = R[n] + R[m];
    else
      R[n] = R[n] + ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
size																															

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)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant alignment = 1;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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			
size																															

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)";
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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	0	0	index_align			Rm						
size																															

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

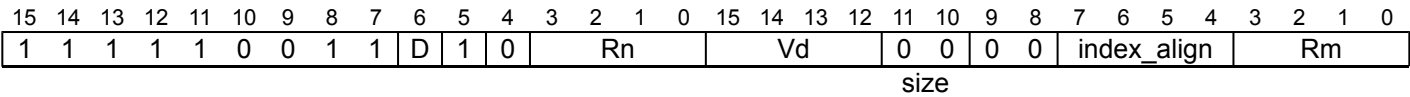
Applies when (Rm != 11x1)

```
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)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant alignment = if index_align<1:0> == '00' then 1 else 4;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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

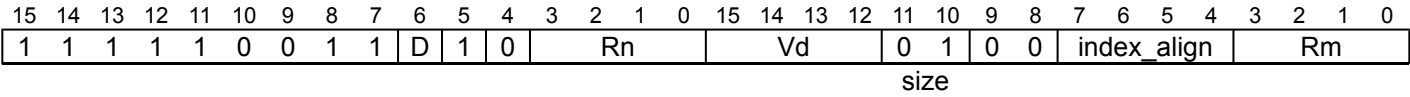
Applies when (Rm != 11x1)

```
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)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant alignment = 1;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2



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)";
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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			
size																															

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)";
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant alignment = if index_align<1:0> == '00' then 1 else 4;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

<c> For encoding A1, A2 and A3: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1, T2 and T3: 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    Elem[D[d],index,8*ebytes] = MemU[address,ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	x	size		align		Rm					
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

```
constant pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
constant inc = if itype == '1001' then 2 else 1;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

## 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

31	30	29	28	27	26	25	24	23	22	21	20	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	1	size		align		Rm					

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

```
constant pairs = 2;  constant inc = 2;
if size == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size);  constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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					
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

```
constant pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
constant inc = if itype == '1001' then 2 else 1;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	1	size		align		Rm					

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

```
constant pairs = 2;  constant inc = 2;
if size == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size);  constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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	RESERVED

<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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for r = 0 to pairs-1
        for e = 0 to elements-1
            Elem[D[d+r], e, 8*bytes] = MemU(address, ebytes);
            Elem[D[d2+r], e, 8*bytes] = MemU(address+ebytes, ebytes);
            address = address + 2*ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						

#### 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;
constant ebytes = 1 << UInt(size);
constant alignment = if a == '0' then 1 else 2*ebytes;
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

## 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						

#### 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;
constant ebytes = 1 << UInt(size);
constant alignment = if a == '0' then 1 else 2*ebytes;
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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 encoding A1: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding T1: 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>&lt;size&gt;</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	<size>	00	8	01	16	10	32	11	RESERVED
size	<size>										
00	8										
01	16										
10	32										
11	RESERVED										
<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 <i>Unaligned data access</i> , 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
                                                         tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    constant integer esize = 8 * ebytes;
    constant bits(esize) element1 = MemU[address, ebytes];
    constant bits(esize) element2 = MemU[address+ebytes, ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						
size																															

#### 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)";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

### 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				0				1				D				1				0				Rn				Vd				0				1				0				1				index_align				Rm			
																																size																															

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

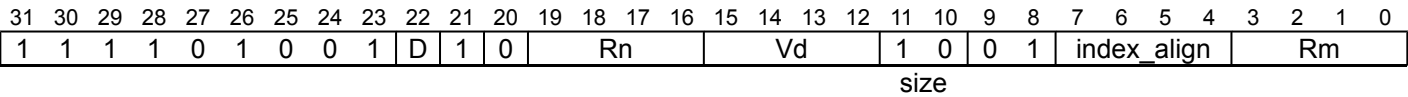
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
constant ebytes = 2;  constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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

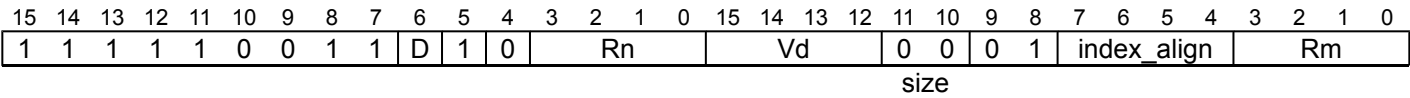
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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

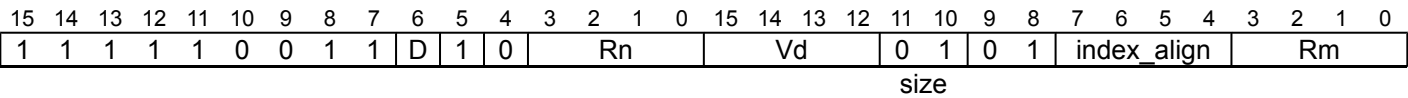
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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



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

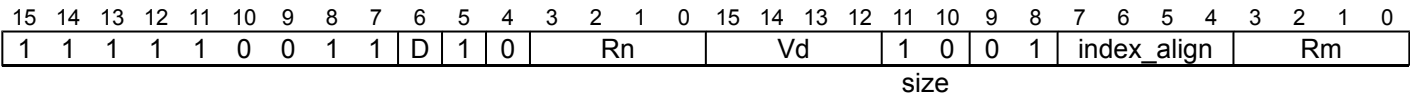
```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
constant ebytes = 2;  constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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)";
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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 encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2 and T3: 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 two SIMD&FP registers holding the element.  
The list must be one of:  
{ <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.  
  
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 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.

: 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    Elem[D[d], index, 8*abytes] = MemU[address,abytes];
    Elem[D[d2], index, 8*abytes] = MemU[address+abytes,abytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*abytes;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 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										
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";
if size == '11' || align<1> == '1' then UNDEFINED;
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align<0> == '0' then 1 else 8;
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

## 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					
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";
if size == '11' || align<1> == '1' then UNDEFINED;
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align<0> == '0' then 1 else 8;
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	RESERVED

- <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 <Rn>, !, and <Rm>, see [Advanced SIMD addressing mode](#).

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckAdvSIMDEnabled();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for e = 0 to elements-1
        Elem[D[d], e, 8*bytes] = MemU[address, bytes];
        Elem[D[d2], e, 8*bytes] = MemU[address+bytes, bytes];
        Elem[D[d3], e, 8*bytes] = MemU[address+2*bytes, bytes];
        address = address + 3*bytes;
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						
																												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;
constant ebytes = 1 << UInt(size);
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### 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						
																												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;
constant ebytes = 1 << UInt(size);
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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 all lanes)*.

Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	RESERVED

<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\(\);
    constant address = R[n];
    constant integer esize = ebytes * 8;
    constant bits(esize) element1 = MemU[address, ebytes];
    constant bits(esize) element2 = MemU[address+ebytes,ebytes];
    constant bits(esize) element3 = MemU[address+2*ebytes,ebytes];

    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    D[d3] = Replicate(element3, 64 DIV esize);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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				size			

#### 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)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

## 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				size			

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

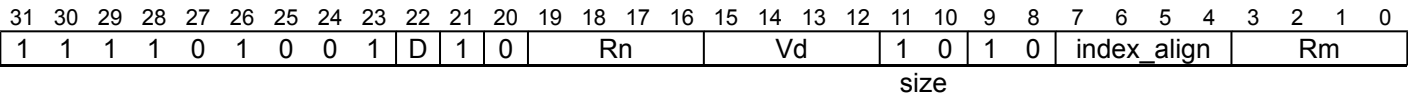
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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

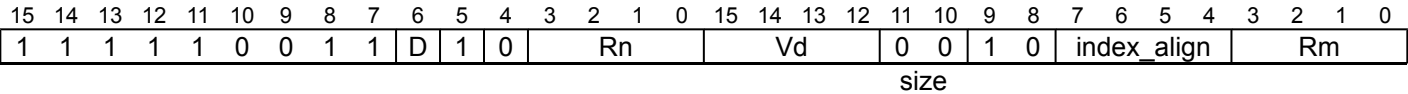
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	0	index_align			Rm						
size																															

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

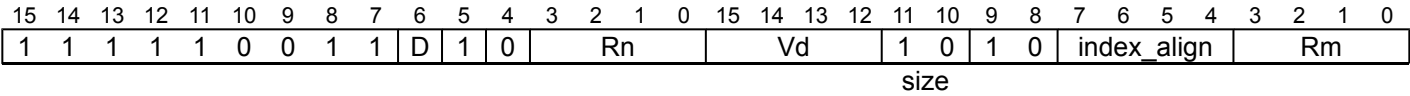
```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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)";
if index_align<1:0> != '00' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2 and T3: 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 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();
    constant address = R[n];
    Elem[D[d], index, 8*bytes] = MemU[address, bytes];
    Elem[D[d2], index, 8*bytes] = MemU[address+bytes, bytes];
    Elem[D[d3], index, 8*bytes] = MemU[address+2*bytes, bytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*bytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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			
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";
if size == '11' then UNDEFINED;
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## 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					
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";
if size == '11' then UNDEFINED;
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	RESERVED

- <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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for e = 0 to elements-1
        Elem[D[d], e, 8*bytes] = MemU[address, bytes];
        Elem[D[d2], e, 8*bytes] = MemU[address+bytes, bytes];
        Elem[D[d3], e, 8*bytes] = MemU[address+2*bytes, bytes];
        Elem[D[d4], e, 8*bytes] = MemU[address+3*bytes, bytes];
        address = address + 4*bytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						

#### 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;
constant ebytes = if size == '11' then 4 else 1 << UInt(size);
integer alignment;
if size == '11' then
    alignment = 16;
else
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

#### 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			

### 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;
constant integer ebytes = if size == '11' then 4 else 1 << UInt(size);
integer alignment;
if size == '11' then
    alignment = 16;
else
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
constant inc = if T == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### 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 encoding A1: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding T1: see <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_LOAD, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    constant integer esize = ebytes * 8;
    constant bits(esome) element1 = MemU[address, ebytes];
    constant bits(esome) element2 = MemU[address+ebytes, ebytes];
    constant bits(esome) element3 = MemU[address+2*ebytes, ebytes];
    constant bits(esome) element4 = MemU[address+3*ebytes, ebytes];
    D[d] = Replicate(element1, 64 DIV esize);
    D[d2] = Replicate(element2, 64 DIV esize);
    D[d3] = Replicate(element3, 64 DIV esize);
    D[d4] = Replicate(element4, 64 DIV esize);
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;

```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 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							
size																															

#### 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)";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### 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			
size																															

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

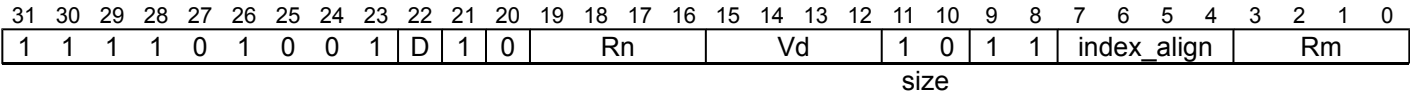
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
constant ebytes = 2;  constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant d3 = d2 + inc;  constant d4 = d3 + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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



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

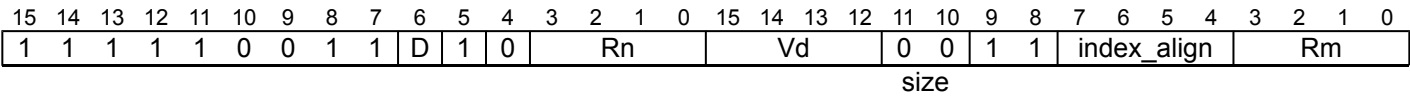
```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<1:0> == '00' then 1 else 4<< UInt(index_align<1:0>);
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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



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)";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## 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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	1	index_align			Rm						
																size															

## 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)";
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## 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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	1	index_align				Rm			
size																															

## 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)";
if index_align<1:0> == '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## 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>	For encoding A1, A2 and A3: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1, T2 and T3: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_LOAD, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    Elem[D[d], index, 8*ebytes] = MemU[address, ebytes];
    Elem[D[d2], index, 8*ebytes] = MemU[address+ebytes, ebytes];
    Elem[D[d3], index, 8*ebytes] = MemU[address+2*ebytes, ebytes];
    Elem[D[d4], index, 8*ebytes] = MemU[address+3*ebytes, ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	imm8<7:1>				0			
cond																imm8<0>															

#### 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>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

#### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

#### 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																															

### 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>
```

```
VLDmia{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = TRUE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(Vd:D); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32); constant regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

### 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	imm8<7:1>								0			
																																imm8<0>			

### 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>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

## Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FLDM*X".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## 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

15	14	13	12	11	10	9	8	7	6	5	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	0	imm8							

## 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>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

## Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VLDR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = TRUE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(Vd:D); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32); constant regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
!	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<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.

Alias Conditions

Alias	Is preferred when
<a href="#">VPOP</a>	<code>P == '0' &amp;&amp; U == '1' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</code>

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    address = if add then R[n] else R[n]-imm32;

    for r = 0 to regs-1
        if single_regs then
            S[d+r] = MemA[address,4];
            address = address+4;
        else
            constant word1 = MemA[address,4];
            constant word2 = MemA[address+4,4];
            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;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.





## 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](#), [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	!= 1111				Vd				1 0		size		imm8							
cond												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = 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							
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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = 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 <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
.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.						
.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.						
<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><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 single-precision scalar or double-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. For the half-precision scalar variant: 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.						

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    CheckVFPEEnabled(TRUE);
    constant base = if n == 15 then Align(PC32,4) else R[n];
    constant address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            H[d] = MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            constant word1 = MemA[address,4];
            constant word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccessType\_ASIMD) then word1:word2 else word2:word1;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

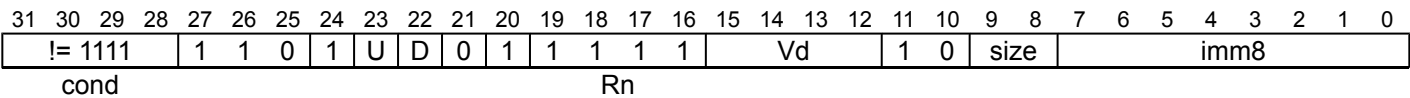
## VLDR (literal)

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



### Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

```
VLDR{<c>}{<q>}.16 <Sd>, <label>

VLDR{<c>}{<q>}.16 <Sd>, [PC, #<+/-><imm>]
```

### Encoding for the Single-precision scalar variant

Applies when (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, <label>

VLDR{<c>}{<q>}.32 <Sd>, [PC, #<+/-><imm>]
```

### Encoding for the Double-precision scalar variant

Applies when (size == 11)

```
VLDR{<c>}{<q>}.64 <Dd>, <label>

VLDR{<c>}{<q>}.64 <Dd>, [PC, #<+/-><imm>]
```

### Decode for all variants of this encoding

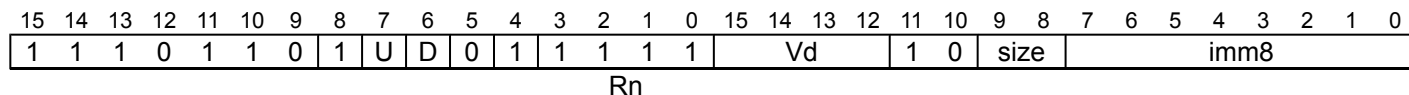
```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = 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



## Encoding for the Half-precision scalar variant (FEAT\_FP16)

Applies when (size == 01)

VLDR{<c>}{<q>}.16 <Sd>, <label>

VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]

## Encoding for the Single-precision scalar variant

Applies when (size == 10)

VLDR{<c>}{<q>}.32 <Sd>, <label>

VLDR{<c>}{<q>}.32 <Sd>, [PC, #{+/-}<imm>]

## Encoding for the Double-precision scalar variant

Applies when (size == 11)

VLDR{<c>}{<q>}.64 <Dd>, <label>

VLDR{<c>}{<q>}.64 <Dd>, [PC, #{+/-}<imm>]

## Decode for all variants of this encoding

```
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = 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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
.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.
.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.
<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”:

U	+/-
0	-
1	+

<imm> For the single-precision scalar or double-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.  
For the half-precision scalar variant: 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.

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();
    CheckVFPEEnabled(TRUE);
    constant base = if n == 15 then Align(PC32,4) else R[n];
    constant address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            H[d] = MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            constant word1 = MemA[address,4];
            constant word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccessType_ASIMD) then word1:word2 else word2:word1;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAX (floating-point)

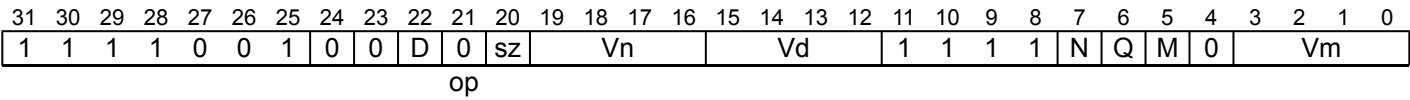
Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

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



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>

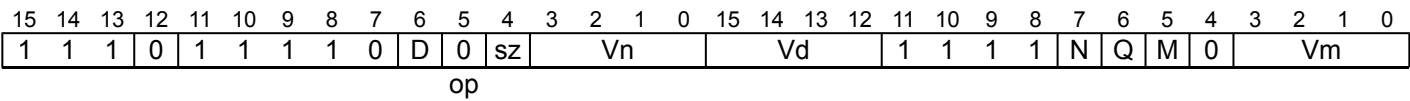
Decode for all variants of this encoding

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;

```

T1



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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.

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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, fpcr);
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	Vn			Vd			0			1	1	0	N	Q	M	0	Vm				
																												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;
if size == '11' then UNDEFINED;
constant maximum = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				
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;
if size == '11' then UNDEFINED;
constant maximum = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Int(Elem[D[n+r],e,esize], unsigned);
            constant op2 = Int(Elem[D[m+r],e,esize], unsigned);
            constant result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAXNM

This instruction determines the floating-point maximum number.

It handles NaNs in consistence with the IEEE754-2008 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			
op																															

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	!= 00		N	0	M	0	Vm			
																	size					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;
constant advsimd = FALSE;
constant maximum = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer 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	1	1	1	0	D	0	sz	Vn			Vd			1	1	1	1	N	Q	M	1	Vm					
op																															

## 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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	!= 00	N	0	M	0	Vm						
																size						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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant advsimd = FALSE;
constant maximum = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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”:

sz	<dt>
0	F32
1	F16

<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.

<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.

<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
    constant FPCR_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r], e, esize];
            constant op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, fpcr);
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, fpcr);
else // VFP instruction
    constant FPCR_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            if maximum then
                H[d] = FPMaxNum(H[n], H[m], fpcr);
            else
                H[d] = FPMinNum(H[n], H[m], fpcr);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], fpcr);
            else
                S[d] = FPMinNum(S[n], S[m], fpcr);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], fpcr);
            else
                D[d] = FPMinNum(D[n], D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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					
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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.

- 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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r],e,esize];
            constant op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, fpcr);
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				
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;
if size == '11' then UNDEFINED;
constant maximum = (op == '0');  constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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				
																												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;
if size == '11' then UNDEFINED;
constant maximum = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Int(Elem[D[n+r],e,esize], unsigned);
            constant op2 = Int(Elem[D[m+r],e,esize], unsigned);
            constant result = if maximum then Max(op1,op2) else Min(op1,op2);
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMINNM

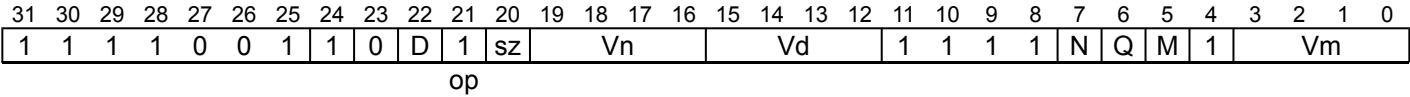
This instruction determines the floating point minimum number.

It handles NaNs in consistence with the IEEE754-2008 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



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

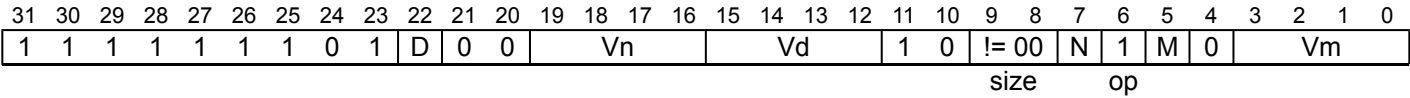
Applies when (Q == 1)

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

A2



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;
constant advsimd = FALSE;
constant maximum = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer 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	1	1	1	0	D	1	sz	Vn			Vd			1	1	1	1	N	Q	M	1	Vm					
op																															

## 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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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	!= 00	N	1	M	0	Vm						
																size						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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant advsimd = FALSE;
constant maximum = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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”:

sz	<dt>
0	F32
1	F16

<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.

<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.

<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
    constant FPCR_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[D[n+r], e, esize];
            constant op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, fpcr);
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, fpcr);
else // VFP instruction
    constant FPCR_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            if maximum then
                H[d] = FPMaxNum(H[n], H[m], fpcr);
            else
                H[d] = FPMinNum(H[n], H[m], fpcr);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], fpcr);
            else
                S[d] = FPMinNum(S[n], S[m], fpcr);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], fpcr);
            else
                D[d] = FPMinNum(D[n], D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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";
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant add = (op == '0'); constant floating_point = (F == '1');
constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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";
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant add = (op == '0'); constant floating_point = (F == '1');
constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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  |
- <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.
- <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".

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant FPCR\_Type fpcr = StandardFPCR();
    constant op2 = Elem[Din[m],index,esize]; constant op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = (if add then FPMul(op1,op2,fpcr)
                             else FPNeg(FPMul(op1,op2,fpcr), fpcr));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, fpcr);
            else
                constant addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																																		

#### 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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE; constant add = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE; constant add = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

## 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					
op																															

## 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

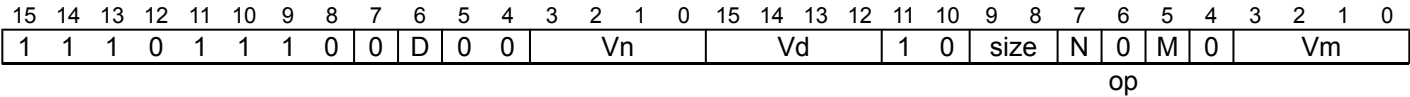
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE; constant add = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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



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;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE; constant add = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.
<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                addend = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], fpcr);
                if !add then addend = FPNeg(addend, fpcr);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, fpcr);
    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                addend16 = FPMul(H[n], H[m], fpcr);
                if !add then addend16 = FPNeg(addend16, fpcr);
                H[d] = FPAdd(H[d], addend16, fpcr);
            when 32
                addend32 = FPMul(S[n], S[m], fpcr);
                if !add then addend32 = FPNeg(addend32, fpcr);
                S[d] = FPAdd(S[d], addend32, fpcr);
            when 64
                addend64 = FPMul(D[n], D[m], fpcr);
                if !add then addend64 = FPNeg(addend64, fpcr);
                D[d] = FPAdd(D[d], addend64, fpcr);

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant add = (op == '0');  constant long_destination = FALSE;
constant unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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																															

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant add = (op == '0'); constant long_destination = FALSE;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	I8
01	I16
10	I32
- <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.
- <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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant product = (Int(Elem[Din[n+r],e,esize],unsigned) *
                          Int(Elem[Din[m+r],e,esize],unsigned));
      constant addend = if add then product else -product;
      if long_destination then
        Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
      else
        Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 for the A1 variant

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant add = (op == '0'); constant floating_point = FALSE;
constant long_destination = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 for the T1 variant

```
VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant add = (op == '0'); constant floating_point = FALSE;
constant long_destination = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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();
    constant FPCR_Type fpcr = StandardFPCR();
    constant op2 = Elem[Din[m],index,esize];    constant op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize];    constant op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = (if add then FPMul(op1,op2,fpcr)
                               else FPNeg(FPMul(op1,op2,fpcr), fpcr));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, fpcr);
            else
                constant addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

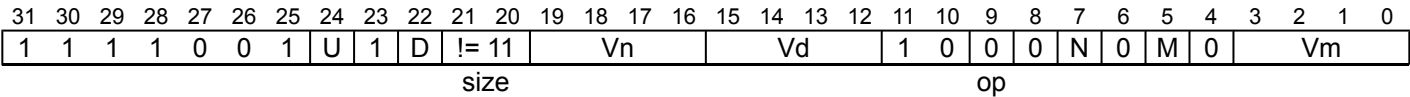
# VMLAL (integer)

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 *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



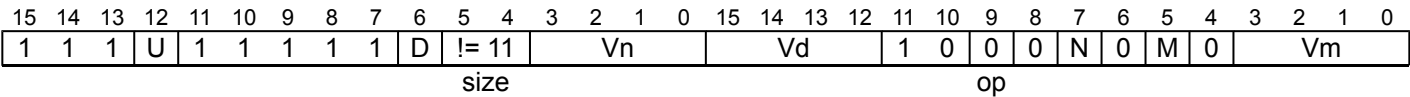
### Encoding for the A1 variant

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');  constant long_destination = TRUE;  constant unsigned = (U == '1');
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);  constant regs = 1;
```

## T1



### Encoding for the T1 variant

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');  constant long_destination = TRUE;  constant unsigned = (U == '1');
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        for e = 0 to elements-1
            constant product = (Int(Elem[Din[n+r],e,esize],unsigned) *
                                Int(Elem[Din[m+r],e,esize],unsigned));
            constant addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



VMLS (by scalar)

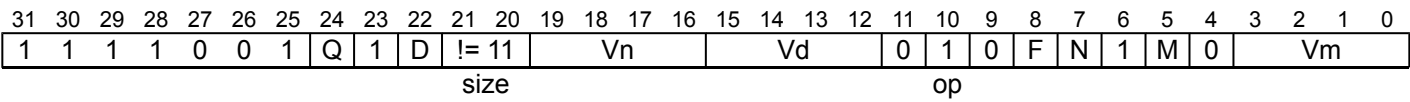
Vector Multiply Subtract multiplies elements of a vector by a scalar, and either subtracts the products from corresponding elements of the destination vector.

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

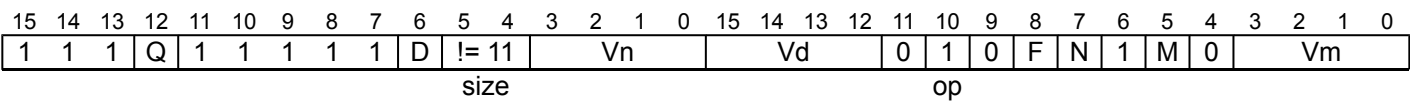
Applies when (Q == 1)

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

Decode for all variants of this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant add = (op == '0'); constant floating_point = (F == '1');
constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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";
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant add = (op == '0'); constant floating_point = (F == '1');
constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

<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.

<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".

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant FPCR\_Type fpcr = StandardFPCR();
    constant op2 = Elem[Din[m],index,esize]; constant op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = (if add then FPMul(op1,op2,fpcr)
                             else FPNeg(FPMul(op1,op2,fpcr), fpcr));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, fpcr);
            else
                constant addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
op																															

#### 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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE; constant add = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE; constant add = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

## 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

op

## 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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE; constant add = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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					
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;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE; constant add = (op == '0');
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

## 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:

sz	<dt>
0	F32
1	F16

- <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.
<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.
<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                addend = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], fpcr);
                if !add then addend = FPNeg(addend, fpcr);
                Elem[D[d+r],e,esize] = FPAdd(Elem[D[d+r],e,esize], addend, fpcr);
    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                addend16 = FPMul(H[n], H[m], fpcr);
                if !add then addend16 = FPNeg(addend16, fpcr);
                H[d] = FPAdd(H[d], addend16, fpcr);
            when 32
                addend32 = FPMul(S[n], S[m], fpcr);
                if !add then addend32 = FPNeg(addend32, fpcr);
                S[d] = FPAdd(S[d], addend32, fpcr);
            when 64
                addend64 = FPMul(D[n], D[m], fpcr);
                if !add then addend64 = FPNeg(addend64, fpcr);
                D[d] = FPAdd(D[d], addend64, fpcr);

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant add = (op == '0');  constant long_destination = FALSE;
constant unsigned = FALSE;  // "Don't care" value: TRUE produces same functionality
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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



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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant add = (op == '0'); constant long_destination = FALSE;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	I8
01	I16
10	I32

<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.

<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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant product = (Int(Elem[Din[n+r],e,esize],unsigned) *
                                Int(Elem[Din[m+r],e,esize],unsigned));
            constant addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:

- The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## VMLS� (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 for the A1 variant

VMLS�{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant add = (op == '0'); constant floating_point = FALSE;
constant long_destination = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 for the T1 variant

VMLS�{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant add = (op == '0'); constant floating_point = FALSE;
constant long_destination = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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();
    constant FPCR_Type fpcr = StandardFPCR();
    constant op2 = Elem[Din[m],index,esize];    constant op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize];    constant op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = (if add then FPMul(op1,op2,fpcr)
                               else FPNeg(FPMul(op1,op2,fpcr), fpcr));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, fpcr);
            else
                constant addend = if add then op1val*op2val else -op1val*op2val;
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
                else
                    Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

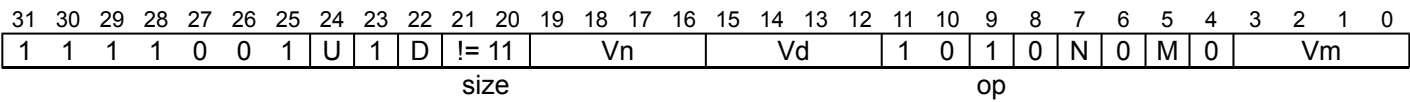
## VMLSL (integer)

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



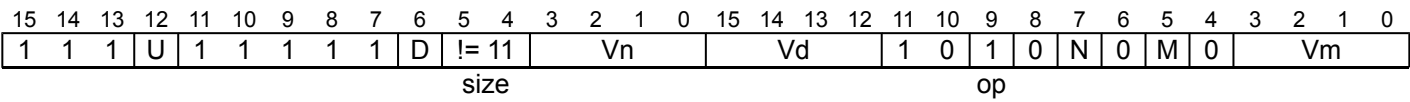
#### Encoding for the A1 variant

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');  constant long_destination = TRUE;  constant unsigned = (U == '1');
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);  constant regs = 1;
```

### T1



#### Encoding for the T1 variant

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');  constant long_destination = TRUE;  constant unsigned = (U == '1');
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);  constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        for e = 0 to elements-1
            constant product = (Int(Elem[Din[n+r],e,esize],unsigned) *
                                Int(Elem[Din[m+r],e,esize],unsigned));
            constant addend = if add then product else -product;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = Elem[Qin[d>>1],e,2*esize] + addend;
            else
                Elem[D[d+r],e,esize] = Elem[Din[d+r],e,esize] + addend;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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					

Encoding for the A1 variant

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = 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					

Encoding for the T1 variant

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32BF16) then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = 2;
```

Assembler Symbols

- <q> See *Standard assembler syntax fields*.
- <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();  
  
constant bits(128) op1 = Q[n>>1];  
constant bits(128) op2 = Q[m>>1];  
constant bits(128) acc = Q[d>>1];  
constant FPCR_Type fpcr = EffectiveFPCR();  
  
Q[d>>1] = BFMatMulAdd(acc, op1, op2, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



## 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;
if cond != '1110' then UNPREDICTABLE;
constant to_arm_register = (op == '1');  constant t = UInt(Rt);  constant n = UInt(Vn:N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

## 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;
if InITBlock() then UNPREDICTABLE;
constant to_arm_register = (op == '1'); constant t = UInt(Rt); constant n = UInt(Vn:N);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

## 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

<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .

## 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>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

```
constant to_arm_register = (op == '1');  constant t = UInt(Rt);  constant n = UInt(Vn:N);  
// Armv8-A removes UNPREDICTABLE for R13  
if t == 15 then UNPREDICTABLE;
```

### T1

15	14	13	12	11	10	9	8	7	6	5	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

```
constant to_arm_register = (op == '1');  constant t = UInt(Rt);  constant n = UInt(Vn:N);  
// Armv8-A removes UNPREDICTABLE for R13  
if t == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<Rt>	Is the general-purpose register that <Sn> will be transferred to or from, encoded in the "Rt" field.
<Sn>	Is the 32-bit name of the SIMD&FP register to be transferred, encoded in the "Vn:N" field.
<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    if to_arm_register then
        R[t] = S[n];
    else
        S[n] = R[t];
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

### 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

```
constant to_arm_registers = (op == '1');  constant t = UInt(Rt);  constant t2 = UInt(Rt2);
constant m = UInt(M:Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

### 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			

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

```
constant to_arm_registers = (op == '1');  constant t = UInt(Rt);  constant t2 = UInt(Rt2);
constant m = UInt(M:Vm);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 || t2 == 15 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

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

<Dm>	Is the 64-bit name of the SIMD&FP register to be transferred, encoded in the "M:Vm" field.
<Rt2>	Is the second general-purpose register that <Dm>[63:32] will be transferred to or from, encoded in the "Rt2" field.
<Rt>	Is the first general-purpose register that <Dm>[31:0] will be transferred to or from, encoded in the "Rt" field.
<c>	See <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .

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];
```

Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

## 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																															

### 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

```
constant t = UInt(Rt);    constant t2 = UInt(Rt2);
constant m = UInt(Vm:M);  constant m2 = m + 1;
constant to_arm_registers = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

### 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			

### 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

```
constant t = UInt(Rt);    constant t2 = UInt(Rt2);
constant m = UInt(Vm:M);  constant m2 = m + 1;
constant to_arm_registers = (op == '1');
if t == 15 || t2 == 15 || m == 31 then UNPREDICTABLE;
if to_arm_registers && t == t2 then UNPREDICTABLE;
```

## 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

<Rt2>	Is the second general-purpose register that <Sm1> will be transferred to or from, encoded in the "Rt2" field.
<Rt>	Is the first general-purpose register that <Sm> will be transferred to or from, encoded in the "Rt" 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>.
<Sm>	Is the 32-bit name of the first SIMD&FP register to be transferred, encoded in the "Vm:M" field.
<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .

## 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];
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:



- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### Encoding for the A1 variant

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

#### Decode for this encoding

```
constant bits(4) opc = opc1:opc2;
if opc == '0x10' then UNDEFINED;
constant integer lsb = LowestSetBit(opc<0,3>);
constant integer esize = 8 << lsb;
constant integer index = UInt(opc<2:lsb>);
constant boolean advsimd = (esize < 32);
constant d = UInt(D:Vd); constant t = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

### T1

15	14	13	12	11	10	9	8	7	6	5	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)	(0)	

#### Encoding for the T1 variant

```
VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>
```

#### Decode for this encoding

```
constant bits(4) opc = opc1:opc2;
if opc == '0x10' then UNDEFINED;
constant integer lsb = LowestSetBit(opc<0,3>);
constant integer esize = 8 << lsb;
constant integer index = UInt(opc<2:lsb>);
constant boolean advsimd = (esize < 32);
constant d = UInt(D:Vd); constant t = UInt(Rt);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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 <a href="#">Standard assembler syntax fields</a> .
<size>	The data size. It must be one of: <b>8</b> Encoded as <code>opc1&lt;1&gt; = 1</code> . [x] is encoded in <code>opc1&lt;0&gt;</code> , <code>opc2</code> . <b>16</b> Encoded as <code>opc1&lt;1&gt; = 0</code> , <code>opc2&lt;0&gt; = 1</code> . [x] is encoded in <code>opc1&lt;0&gt;</code> , <code>opc2&lt;1&gt;</code> . <b>32</b> Encoded as <code>opc1&lt;1&gt; = 0</code> , <code>opc2 = 0b00</code> . [x] is encoded in <code>opc1&lt;0&gt;</code> . <b>omitted</b> 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);
    Elem[D[d],index,esize] = R[t]<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant bits(esize) imm = VFPEExpandImm(imm4H:imm4L, esize);
constant advsimd = FALSE;
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer regs = 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant integer esize = 8 << UInt(size);
constant bits(esize) imm = VFPExpandImm(imm4H:imm4L, esize);
constant advsimd = FALSE;
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer regs = 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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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)

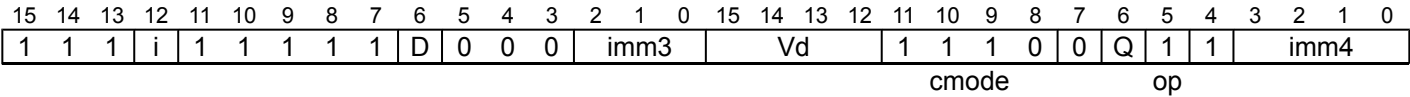
```
VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>
```

#### Decode for all variants of this encoding

```
if op == '0' && cmode<0> == '1' && cmode<3:2> != '11' then SEE "VORR (immediate)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = if Q == '0' then 1 else 2;
```

### T5





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)";
if op == '1' && cmode != '1110' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 64;
constant bits(esize) imm = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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 encoding A1, A3, A4 and A5: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding A2, T1, T2, T3, T4 and T5: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> The data type, encoded in “cmode”:

cmode	<dt>
110x	I32
1110	I8
1111	F32
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <imm> For encoding A1, A3, A4, A5, T1, T3, T4 and T5: 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 encoding A2 and T2: 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](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if esize <= 32 then
        S[d] = ZeroExtend(imm, 32);
    else
        for r = 0 to regs-1
            D[d+r] = ZeroExtend(imm, 64);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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																size															

#### 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;
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant single_register = (size == '10');
constant advsimd = 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			
																size															

#### 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;
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant single_register = (size == '10');
constant advsimd = FALSE;
```

## 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.
<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(); CheckAdvSIMDOrVFPEntered(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        D[d] = D[m];
```

## Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# 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			

### 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	0	1	N	Q	M	1	Vm					

### 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> An optional data type. <dt> must not be F64, but it is otherwise ignored.
- <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.
- <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.

Operation

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

## 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																															

#### Encoding for the A1 variant

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

#### Decode for this encoding

```
constant bits(4) opc = opc1:opc2;
if (U:opc) IN {'10x00', 'x0x10'} then UNDEFINED;
constant integer lsb = LowestSetBit(opc<0,3>);
constant integer esize = 8 << lsb;
constant integer index = UInt(opc<2:lsb>);
constant boolean advsimd = (esize < 32);
constant boolean unsigned = (U == '1');
constant t = UInt(Rt); constant n = UInt(N:Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

### T1

15	14	13	12	11	10	9	8	7	6	5	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)		

#### Encoding for the T1 variant

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

#### Decode for this encoding

```
constant bits(4) opc = opc1:opc2;
if (U:opc) IN {'10x00', 'x0x10'} then UNDEFINED;
constant integer lsb = LowestSetBit(opc<0,3>);
constant integer esize = 8 << lsb;
constant integer index = UInt(opc<2:lsb>);
constant boolean advsimd = (esize < 32);
constant boolean unsigned = (U == '1');
constant t = UInt(Rt); constant n = UInt(N:Vn);
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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

## Assembler Symbols

<c>	See <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	The data type. It must be one of: <b>S8</b> Encoded as $U = 0$ , $opc1<1> = 1$ . [x] is encoded in $opc1<0>$ , $opc2$ . <b>S16</b> Encoded as $U = 0$ , $opc1<1> = 0$ , $opc2<0> = 1$ . [x] is encoded in $opc1<0>$ , $opc2<1>$ . <b>U8</b> Encoded as $U = 1$ , $opc1<1> = 1$ . [x] is encoded in $opc1<0>$ , $opc2$ . <b>U16</b> Encoded as $U = 1$ , $opc1<1> = 0$ , $opc2<0> = 1$ . [x] is encoded in $opc1<0>$ , $opc2<1>$ . <b>32</b> Encoded as $U = 0$ , $opc1<1> = 0$ , $opc2 = 0b00$ . [x] is encoded in $opc1<0>$ . <b>omitted</b> 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(Elem[D[n], index, esize], 32);
    else
        R\[t\] = SignExtend(Elem[D[n], index, esize], 32);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.



# 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	!= 000			0	0	0	Vd			1 0 1 0			0	0	M	1	Vm					

imm3H

### Encoding for the A1 variant

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

### Decode for this encoding

```
if imm3H == '000' then SEE "Related encodings";
if ! imm3H IN {'001', '010', '100'} then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
constant esize = 8 << HighestSetBitNZ(imm3H);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant unsigned = (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	!= 000			0	0	0	Vd			1 0 1 0			0	0	M	1	Vm					

imm3H

### Encoding for the T1 variant

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

### Decode for this encoding

```
if imm3H == '000' then SEE "Related encodings";
if ! imm3H IN {'001', '010', '100'} then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
constant esize = 8 << HighestSetBitNZ(imm3H);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant unsigned = (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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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
        constant result = Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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

31	30	29	28	27	26	25	24	23	22	21	20	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				

Encoding for the A1 variant

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = 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	0	0	0	M	0	Vm				

Encoding for the T1 variant

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

Decode for this encoding

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);
```

Assembler Symbols

- <c>

For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        Elem[D[d],e,esize] = Elem[Qin[m>>1],e,2*esize]<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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*, *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 (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
1	1	1	1	1	1	1	0	1	D	1	1	0	0	0	0				Vd		1	0	1	0	0	1	M	0			Vm

### Encoding for the A1 variant

VMOVX{<q>}.F16 <Sd>, <Sm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant d = UInt(Vd:D); constant m = UInt(Vm:M);
```

## 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	1	1	1	1	0	1	D	1	1	0	0	0	0				Vd		1	0	1	0	0	1	M	0			Vm

### Encoding for the T1 variant

VMOVX{<q>}.F16 <Sd>, <Sm>

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant d = UInt(Vd:D); constant m = 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 <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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    H[d] = S[m]<31:16>;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### Encoding for the A1 variant

VMRS{<c>}{<q>} <Rt>, <spec\_reg>

### Decode for this encoding

```
constant t = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UNPREDICTABLE;
```

## 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)	1	(0)	(0)	(0)	(0)

### Encoding for the T1 variant

VMRS{<c>}{<q>} <Rt>, <spec\_reg>

### Decode for this encoding

```
constant t = UInt(Rt);
if ! reg IN {'000x', '0101', '011x', '1000'} then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 && reg != '0001' then UNPREDICTABLE;
```

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;
    elsif 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
```





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																															

Encoding for the A1 variant

VMSR{<c>}{<q>} <spec\_reg>, <Rt>

Decode for this encoding

```
constant t = UInt(Rt);
if reg != '000x' && reg != '1000' then
    constant Constraint c = ConstrainUnpredictable(Unpredictable VMSR);
    assert c IN {Constraint UNDEF, Constraint NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            ExecuteAsNOP();
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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)

Encoding for the T1 variant

```
VMSR{<c>}{<q>} <spec_reg>, <Rt>
```

Decode for this encoding

```
constant t = UInt(Rt);
if reg != '000x' && reg != '1000' then
    constant Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            ExecuteAsNOP();
// Armv8-A removes UNPREDICTABLE for R13
if t == 15 then UNPREDICTABLE;
```

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];
    elsif PSTATE.EL == ELO 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
            when '1000'
                FPEXC = R[t];
            otherwise
                Unreachable(); // Dealt with above or in encoding-specific pseudocode
```



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";
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant floating_point = (F == '1'); constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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";
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if size == '00' || (F == '1' && size == '01' && !IsFeatureImplemented(FEAT_FP16)) then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant floating_point = (F == '1'); constant long_destination = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

<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.

<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.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant FPCR\_Type fpcr = StandardFPCR();
    constant op2 = Elem[Din[m],index,esize]; constant op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, fpcr);
            else
                if long_destination then
                    Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
                else
                    Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	1	0	1	N	Q	M	1	Vm			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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			

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 UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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					

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 UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding A2, T1 and T2: 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>&lt;dt&gt;</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						
<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.						
<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.						
<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], fpcr);
    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                H[d] = FPMul(H[n], H[m], fpcr);
            when 32
                S[d] = FPMul(S[n], S[m], fpcr);
            when 64
                D[d] = FPMul(D[n], D[m], fpcr);

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant polynomial = (op == '1'); constant long_destination = FALSE;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
constant polynomial = (op == '1'); constant long_destination = FALSE;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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:size”:

op	size	<dt>
0	00	I8
0	01	I16
0	10	I32
1	00	P8

<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.

<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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
      constant op2 = Elem[Din[m+r],e,esize]; constant op2val = Int(op2, unsigned);
      bits(2 * esize) product;
      if polynomial then
        product = PolynomialMult(op1,op2);
      else
        product = (op1val*op2val)<2*esize-1:0>;
      if long_destination then
        Elem[Q[d>>1],e,2*esize] = product;
      else
        Elem[D[d+r],e,esize] = product<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	1	0	N	1	M	0		Vm	
										size																					

### Encoding for the A1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE; constant floating_point = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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				1	0	1	0	N	1	M	0	Vm				
										size																					

### Encoding for the T1 variant

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant long_destination = TRUE; constant floating_point = FALSE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant regs = 1;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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 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();
  constant FPCR\_Type fpcr = StandardFPCR();
  constant op2 = Elem[Din[m],index,esize]; constant op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
      if floating_point then
        Elem[D[d+r],e,esize] = FPMul(op1, op2, fpcr);
      else
        if long_destination then
          Elem[Q[d>>1],e,2*esize] = (op1val*op2val)<2*esize-1:0>;
        else
          Elem[D[d+r],e,esize] = (op1val*op2val)<esize-1:0>;
```

## 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 [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		1	1	op	0	N	0	M	0		Vm
size																															

#### Encoding for the A1 variant

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if op == '1' && (U == '1' || size == '01') then UNDEFINED;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then UNDEFINED;
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant polynomial = (op == '1'); constant long_destination = TRUE;
constant integer esize = if polynomial && size == '10' then 64 else 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 for the T1 variant

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if op == '1' && (U == '1' || size == '01') then UNDEFINED;
if op == '1' && size == '10' && InITBlock() then UNPREDICTABLE;
if op == '1' && size == '10' && !IsFeatureImplemented(FEAT_PMULL) then UNPREDICTABLE;
if Vd<0> == '1' then UNDEFINED;
constant unsigned = (U == '1'); constant polynomial = (op == '1'); constant long_destination = TRUE;
constant integer esize = if polynomial && size == '10' then 64 else 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 64 DIV esize; constant regs = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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
        for e = 0 to elements-1
            constant op1 = Elem[Din[n+r],e,esize]; constant op1val = Int(op1, unsigned);
            constant op2 = Elem[Din[m+r],e,esize]; constant op2val = Int(op2, unsigned);
            bits(2 * esize) product;
            if polynomial then
                product = PolynomialMult(op1,op2);
            else
                product = (op1val*op2val)<2*esize-1:0>;
            if long_destination then
                Elem[Q[d>>1],e,2*esize] = product;
            else
                Elem[D[d+r],e,esize] = product<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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																															

#### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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															

#### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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

### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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

### 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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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																															

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

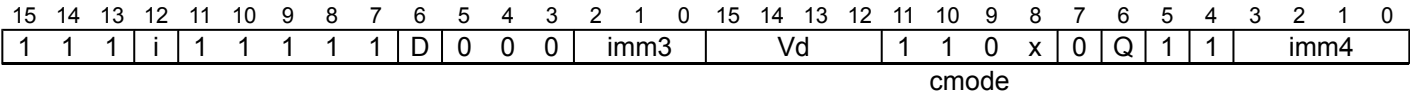
Applies when (Q == 1)

```
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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = if Q == '0' then 1 else 2;
```

T3



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";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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 encoding A1, A2 and A3: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1, T2 and T3: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <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](#).

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(imm64);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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					

#### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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					

#### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.



For encoding T1: 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.

<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.

<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = NOT(D[m+r]);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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	1	Q	M	0	Vm				

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant advsimd = TRUE;  constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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

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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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				

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;
if F == '1' && ((size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size == '00') then
    UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant advsimd = TRUE;  constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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					

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;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant boolean floating_point = boolean UNKNOWN;
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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

<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.

<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.

<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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                if floating_point then
                    Elem[D[d+r],e,esize] = FPNeg(Elem[D[m+r],e,esize], fpcr);
                else
                    constant result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
            else
                // VFP instruction
                constant FPCR\_Type fpcr = EffectiveFPCR();
                case esize of
                    when 16 H[d] = FPNeg(H[m], fpcr);
                    when 32 S[d] = FPNeg(S[m], fpcr);
                    when 64 D[d] = FPNeg(D[m], fpcr);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check and is operating only on integer vector elements, then the following apply:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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.

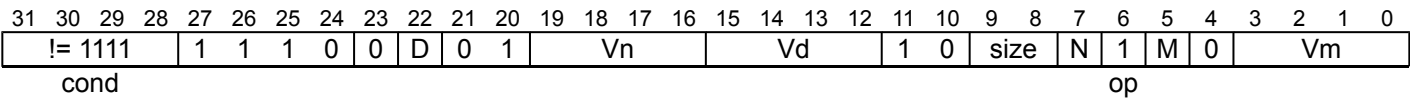
## Note

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



### 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

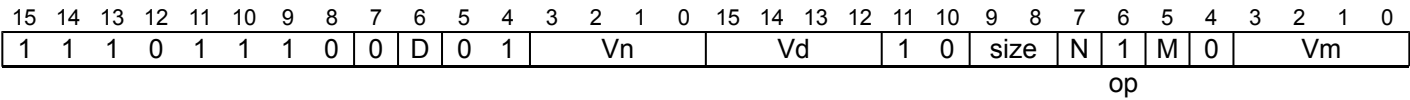
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant vtype = if op == '1' then VFPNegMul VNMLA else VFPNegMul VNMLS;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant product16 = FPMul(H[n], H[m], fpcr);
            case vtype of
                when VFPNegMul\_VNMLA H[d] = FPAdd(FPNeg(H[d], fpcr), FPNeg(product16, fpcr), fpcr);
                when VFPNegMul\_VNMLS H[d] = FPAdd(FPNeg(H[d], fpcr), product16, fpcr);
                when VFPNegMul\_VNMUL H[d] = FPNeg(product16, fpcr);
            when 32
                constant product32 = FPMul(S[n], S[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA S[d] = FPAdd(FPNeg(S[d], fpcr), FPNeg(product32, fpcr), fpcr);
                    when VFPNegMul\_VNMLS S[d] = FPAdd(FPNeg(S[d], fpcr), product32, fpcr);
                    when VFPNegMul\_VNMUL S[d] = FPNeg(product32, fpcr);
            when 64
                constant product64 = FPMul(D[n], D[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA D[d] = FPAdd(FPNeg(D[d], fpcr), FPNeg(product64, fpcr), fpcr);
                    when VFPNegMul\_VNMLS D[d] = FPAdd(FPNeg(D[d], fpcr), product64, fpcr);
                    when VFPNegMul\_VNMUL D[d] = FPNeg(product64, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

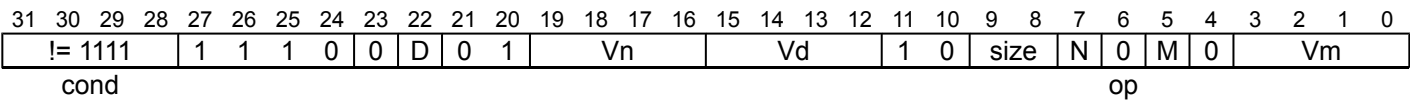
VNMLS

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](#), [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



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

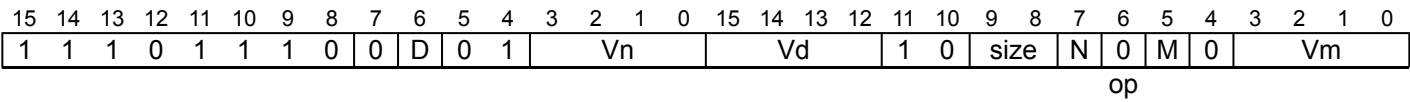
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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



**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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant product16 = FPMul(H[n], H[m], fpcr);
            case vtype of
                when VFPNegMul\_VNMLA H[d] = FPAdd(FPNeg(H[d], fpcr), FPNeg(product16, fpcr), fpcr);
                when VFPNegMul\_VNMLS H[d] = FPAdd(FPNeg(H[d], fpcr), product16, fpcr);
                when VFPNegMul\_VNMUL H[d] = FPNeg(product16, fpcr);
            when 32
                constant product32 = FPMul(S[n], S[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA S[d] = FPAdd(FPNeg(S[d], fpcr), FPNeg(product32, fpcr), fpcr);
                    when VFPNegMul\_VNMLS S[d] = FPAdd(FPNeg(S[d], fpcr), product32, fpcr);
                    when VFPNegMul\_VNMUL S[d] = FPNeg(product32, fpcr);
            when 64
                constant product64 = FPMul(D[n], D[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA D[d] = FPAdd(FPNeg(D[d], fpcr), FPNeg(product64, fpcr), fpcr);
                    when VFPNegMul\_VNMLS D[d] = FPAdd(FPNeg(D[d], fpcr), product64, fpcr);
                    when VFPNegMul\_VNMUL D[d] = FPNeg(product64, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant vtype = VFPNegMul\_VNMUL;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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				

**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;
if size == '01' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant vtype = VFPNegMul_VNMUL;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = 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 <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.
<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);
    constant FPCR\_Type fpcr = EffectiveFPCR();
    case esize of
        when 16
            constant product16 = FPMul(H[n], H[m], fpcr);
            case vtype of
                when VFPNegMul\_VNMLA H[d] = FPAdd(FPNeg(H[d], fpcr), FPNeg(product16, fpcr), fpcr);
                when VFPNegMul\_VNMLS H[d] = FPAdd(FPNeg(H[d], fpcr), product16, fpcr);
                when VFPNegMul\_VNMUL H[d] = FPNeg(product16, fpcr);
            when 32
                constant product32 = FPMul(S[n], S[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA S[d] = FPAdd(FPNeg(S[d], fpcr), FPNeg(product32, fpcr), fpcr);
                    when VFPNegMul\_VNMLS S[d] = FPAdd(FPNeg(S[d], fpcr), product32, fpcr);
                    when VFPNegMul\_VNMUL S[d] = FPNeg(product32, fpcr);
            when 64
                constant product64 = FPMul(D[n], D[m], fpcr);
                case vtype of
                    when VFPNegMul\_VNMLA D[d] = FPAdd(FPNeg(D[d], fpcr), FPNeg(product64, fpcr), fpcr);
                    when VFPNegMul\_VNMLS D[d] = FPAdd(FPNeg(D[d], fpcr), product64, fpcr);
                    when VFPNegMul\_VNMUL D[d] = FPNeg(product64, fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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>

### 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																															

#### Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>}, {<Dd>}, #<imm>

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>

## T1

15	14	13	12	11	10	9	8	7	6	5	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																															

### 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>

## 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																															

### Encoding for the 64-bit SIMD vector variant

Applies when (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>, } <Dd>, #<imm>

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>

## Assembler Symbols

- <c> For encoding A1 and A2: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1 and T2: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <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](#).

## Operation

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

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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 *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	1	Vn				Vd				0				0	0	1	N	Q	M	1	Vm			

### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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			

### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR NOT(D[m+r]);
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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																															

#### 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)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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															

#### 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)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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																															

### 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)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = 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																															

### 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)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
constant imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
constant d = UInt(D:Vd); constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

- <c> For encoding A1 and A2: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1 and T2: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Qd> Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>\*2.
- <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](#).

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
        D[d+r] = D[d+r] OR imm64;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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			

#### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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					

#### 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;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```



Assembler Symbols

- <c>For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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.

Alias Conditions

Alias	Is preferred when
<a href="#">VMOV (register, SIMD)</a>	N:Vn == M:Vm
<a href="#">VRSHR (zero)</a>	Never
<a href="#">VSHR (zero)</a>	Never

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        D[d+r] = D[n+r] OR D[m+r];
```

Operational information

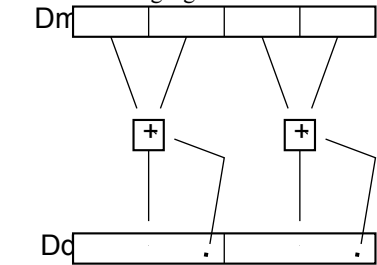
- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> 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
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	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.
- <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();  CheckAdvSIMDEnabled();
    constant h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            constant op1 = Elem[D[m+r],2*e,esize];
            constant op2 = Elem[D[m+r],2*e+1,esize];
            constant result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = Elem[D[d+r],e,2*esize] + result;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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			

#### Encoding for the A1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if Q == '1' then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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			

#### Encoding for the T1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if Q == '1' then UNDEFINED;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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 encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .

<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant FPCR\_Type fpcr = StandardFPCR();
    bits(64) dest;
    constant h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]    = FPAdd(Elem[D[n],2*e,esize], Elem[D[n],2*e+1,esize], fpcr);
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize], fpcr);

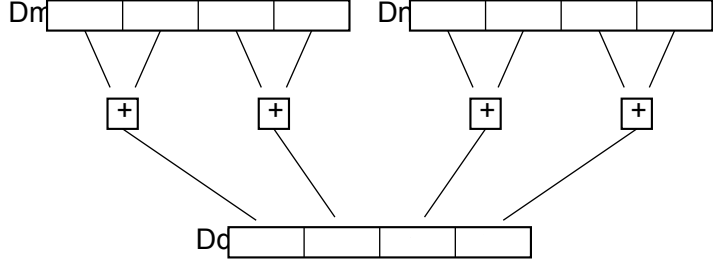
    D[d] = dest;
```

## 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			

#### Encoding for the A1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' || Q == '1' then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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				

#### Encoding for the T1 variant

VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' || Q == '1' then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
```

### Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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();
    bits(64) dest;
    constant h = elements DIV 2;

    for e = 0 to h-1
        Elem[dest,e,esize]    = Elem[D[n],2*e,esize] + Elem[D[n],2*e+1,esize];
        Elem[dest,e+h,esize] = Elem[D[m],2*e,esize] + Elem[D[m],2*e+1,esize];

    D[d] = dest;
```

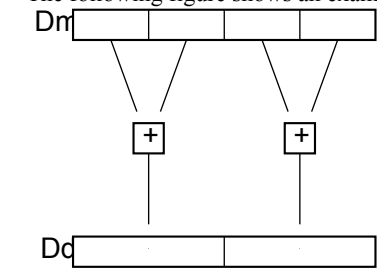
Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

# 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					

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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			

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant unsigned = (op == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> 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
0	11	RESERVED
1	00	U8
1	01	U16
1	10	U32
1	11	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.
- <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();  CheckAdvSIMDEnabled();
    constant h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            constant op1 = Elem[D[m+r],2*e,esize];
            constant op2 = Elem[D[m+r],2*e+1,esize];
            constant result = Int(op1, unsigned) + Int(op2, unsigned);
            Elem[D[d+r],e,2*esize] = result<2*esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	M	0	Vm			
op																															

### Encoding for the A1 variant

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

### Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	1	1	N	0	M	0	Vm					
op																															

### Encoding for the T1 variant

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

### Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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”:

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();
    bits(64) dest;
    constant FPCR\_Type fpcr = StandardFPCR();
    constant h = elements DIV 2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize]; op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = (if maximum then FPCMax(op1,op2,fpcr)
                           else FPCMin(op1,op2,fpcr));
        op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = (if maximum then FPCMax(op1,op2,fpcr)
                              else FPCMin(op1,op2,fpcr));

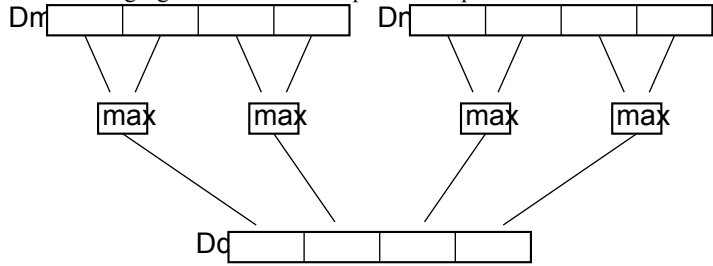
    D[d] = dest;

```

## 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							
																								op							

#### Encoding for the A1 variant

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then UNDEFINED;
constant maximum = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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				
op																															

#### Encoding for the T1 variant

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

#### Decode for this encoding

```
if size == '11' then UNDEFINED;
constant maximum = (op == '0'); constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
```

### Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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();
    bits(64) dest;
    constant h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

## 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	M	0	Vm			
op																															

### Encoding for the A1 variant

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

### Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	1	sz	Vn			Vd			1	1	1	1	N	0	M	0	Vm					
op																															

### Encoding for the T1 variant

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

### Decode for this encoding

```
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant maximum = (op == '0');
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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”:

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();
    bits(64) dest;
    constant FPCR\_Type fpcr = StandardFPCR();
    constant h = elements DIV 2;

    for e = 0 to h-1
        op1 = Elem[D[n],2*e,esize]; op2 = Elem[D[n],2*e+1,esize];
        Elem[dest,e,esize] = (if maximum then FPCMax(op1,op2,fpcr)
                           else FPCMin(op1,op2,fpcr));
        op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = (if maximum then FPCMax(op1,op2,fpcr)
                              else FPCMin(op1,op2,fpcr));

    D[d] = dest;

```

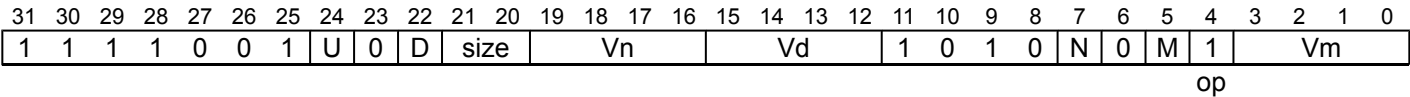
VPMIN (integer)

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



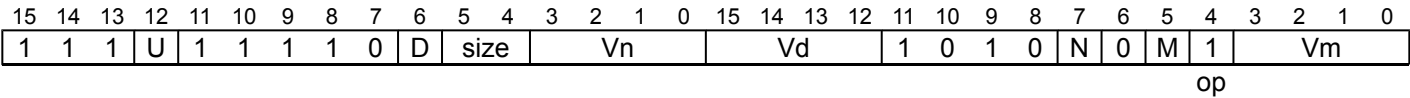
Encoding for the A1 variant

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' then UNDEFINED;
constant maximum = (op == '0');  constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
```

T1



Encoding for the T1 variant

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

Decode for this encoding

```
if size == '11' then UNDEFINED;
constant maximum = (op == '0');  constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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();
    bits(64) dest;
    constant h = elements DIV 2;

    for e = 0 to h-1
        op1 = Int(Elem[D[n],2*e,esize], unsigned);
        op2 = Int(Elem[D[n],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e,esize] = result<esize-1:0>;
        op1 = Int(Elem[D[m],2*e,esize], unsigned);
        op2 = Int(Elem[D[m],2*e+1,esize], unsigned);
        result = if maximum then Max(op1,op2) else Min(op1,op2);
        Elem[dest,e+h,esize] = result<esize-1:0>;

    D[d] = dest;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	imm8<7:1>					0	
cond				P		U	W		Rn															imm8<0>								

### 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		Rn																						

### 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	imm8<7:1>				0			
P							U		W			Rn											imm8<0>								

### Encoding for the Increment After variant

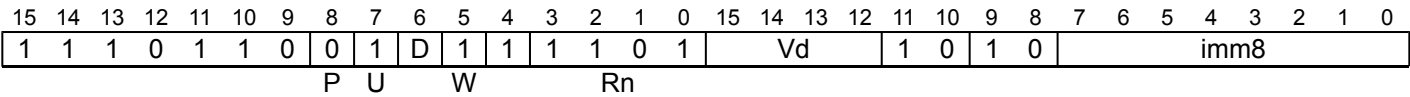
VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VLDM{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2



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 [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.
- <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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

# 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	imm8<7:1>				0			
cond				P		U	W		Rn				imm8<0>																		

### 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		Rn																							

### 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	imm8<7:1>				0				
							P		U				Rn				imm8<0>																	

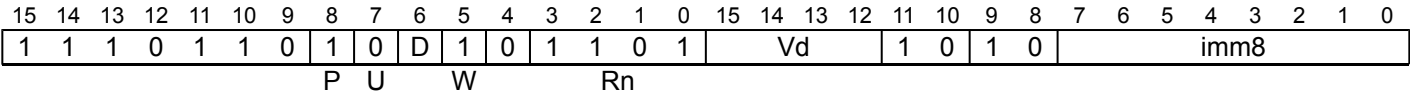
### Encoding for the Decrement Before variant

VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

VSTMDB{<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.



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 *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.
- <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

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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

31	30	29	28	27	26	25	24	23	22	21	20	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	1	0	Q	M	0	Vm					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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	1	0	Q	M	0		Vm				

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c>For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	S8
01	S16
10	S32
11	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.
- <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(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant result = Abs(SInt(Elem[D[m+r],e,esize]));
      boolean sat;
      (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
      if sat then FPSCR.QC = '1';
```

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

31	30	29	28	27	26	25	24	23	22	21	20	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	N	Q	M	1		Vm

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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	1		Vm

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```



Assembler Symbols

- <c>For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant sum = (Int(Elem[D[n+r],e,esize], unsigned) +
                     Int(Elem[D[m+r],e,esize], unsigned));
      boolean sat;
      (Elem[D[d+r],e,esize], sat) = SatQ(sum, esize, unsigned);
      if sat then FPSCR.QC = '1';
```

## 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				
size											op																				

#### Encoding for the A1 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = FALSE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

### 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						
size											op																				

#### Encoding for the A2 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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	0	1	1	1	1	1	D	!= 11	Vn			Vd			1	0	0	1	N	0	M	0	Vm						
size																op															

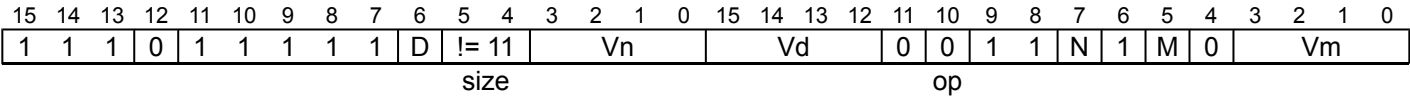
Encoding for the T1 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = FALSE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

T2



Encoding for the T2 variant

```
VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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 encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.  
For encoding A2 and T2: 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();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        constant op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        constant (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        integer result;
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        boolean sat2;
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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						
size											op																				

#### Encoding for the A1 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = FALSE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

### 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				
size											op																				

#### Encoding for the A2 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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	0	1	1	1	1	1	D	!= 11	Vn			Vd			1	0	1	1	N	0	M	0	Vm						
size																op															

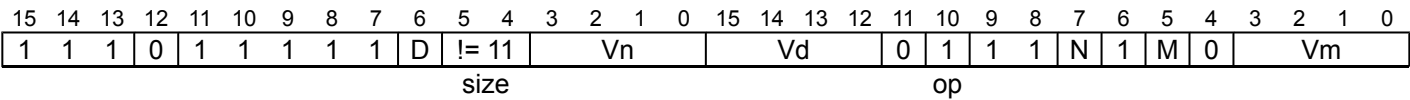
Encoding for the T1 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = FALSE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

T2



Encoding for the T2 variant

```
VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant add = (op == '0');
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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 encoding A1 and T1: is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.  
For encoding A2 and T2: 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();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        constant op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal  $-(2^{(esize-1)})$ 
        constant (product, sat1) = SignedSatQ(2*op1*op2, 2*esize);
        integer result;
        if add then
            result = SInt(Elem[Qin[d>>1],e,2*esize]) + SInt(product);
        else
            result = SInt(Elem[Qin[d>>1],e,2*esize]) - SInt(product);
        boolean sat2;
        (Elem[Q[d>>1],e,2*esize], sat2) = SignedSatQ(result, 2*esize);
        if sat1 || sat2 then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *VQORDMULH*. 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		

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;
if size == '00' || size == '11' then UNDEFINED;
constant scalar_form = FALSE;  constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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																															



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";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer      m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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	0	1	1	1	1	0	D	size	Vn				Vd				1	0	1	1	N	Q	M	0	Vm				

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;
if size == '00' || size == '11' then UNDEFINED;
constant scalar_form = FALSE;  constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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	Q	1	1	1	1	1	D	!= 11				Vn				Vd				1	1	0	0	N	1	M	0	Vm			

size

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";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer      m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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.
- <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.
- <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[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".
- <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();
    integer op2;
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            constant op1 = SInt(Elem[D[n+r],e,esize]);
            // The following only saturates if both op1 and op2 equal  $-(2^{(esize-1)})$ 
            constant (result, sat) = SignedSatQ((2*op1*op2) >> esize, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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							
										size																					

#### Encoding for the A1 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant scalar_form = FALSE; constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

### 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						
										size																					

#### Encoding for the A2 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

#### Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant scalar_form = TRUE; constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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	0	1	1	1	1	1	D	!= 11	Vn			Vd			1	1	0	1	N	0	M	0	Vm						
										size																					

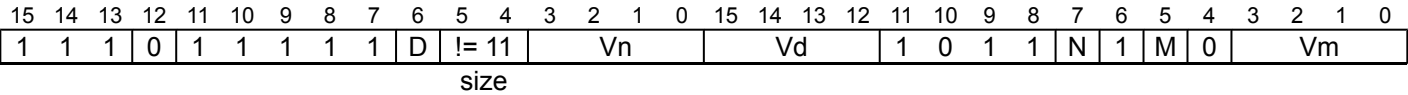
Encoding for the T1 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant scalar_form = FALSE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant m = UInt(M:Vm);
constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant integer index = integer UNKNOWN;
```

T2



Encoding for the T2 variant

```
VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>
```

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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[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".
- <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();
    integer op2;
    if scalar_form then op2 = SInt(Elem[Din[m],index,esize]);
    for e = 0 to elements-1
        if !scalar_form then op2 = SInt(Elem[Din[m],e,esize]);
        constant op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal  $-(2^{(esize-1)})$ 
        constant (product, sat) = SignedSatQ(2*op1*op2, 2*esize);
        Elem[Q[d>>1],e,2*esize] = product;
        if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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 [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](#).

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		

#### 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";
if size == '11' || Vm<0> == '1' then UNDEFINED;
constant src_unsigned = (op == '11'); constant dest_unsigned = (op<0> == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = 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		

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";
if size == '11' || Vm<0> == '1' then UNDEFINED;
constant src_unsigned = (op == '11'); constant dest_unsigned = (op<0> == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <dt> For the signed result variant: is the data type for the elements of the operand, encoded in “op<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	11	RESERVED

For the unsigned result variant: 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
    constant operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
    boolean sat;
    (Elem[D[d],e,esize], sat) = SatQ(operand, esize, dest_unsigned);
    if sat then FPSCR.QC = '1';
```



# 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

31	30	29	28	27	26	25	24	23	22	21	20	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	1	1	Q	M	0	Vm					

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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	1	1	Q	M	0		Vm					

### 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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	S8
01	S16
10	S32
11	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.
- <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(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant result = -SInt(Elem[D[m+r],e,esize]);
      boolean sat;
      (Elem[D[d+r],e,esize], sat) = SignedSatQ(result, esize);
      if sat then FPSCR.QC = '1';
```

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 *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  
(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		

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant add = TRUE; constant scalar_form = FALSE; constant esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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																															

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;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant add = TRUE; constant scalar_form = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = if size == '01' then UInt(M:Vm<3>) else UInt(M);
```

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				

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

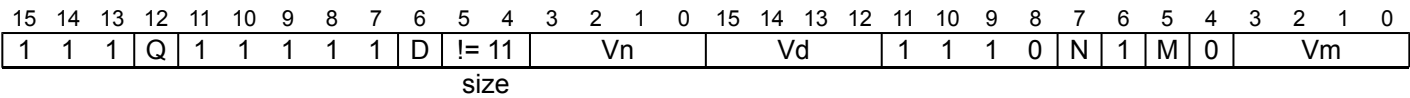
```
if !IsFeatureImplemented(FEAT_RDM) then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant add = TRUE; constant scalar_form = FALSE; constant esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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)



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;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant add = TRUE; constant scalar_form = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = if size == '01' then UInt(M:Vm<3>) else UInt(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.

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”: <table><tr><th>size</th><th>&lt;dt&gt;</th></tr><tr><td>01</td><td>S16</td></tr><tr><td>10</td><td>S32</td></tr></table>	size	<dt>	01	S16	10	S32
size	<dt>						
01	S16						
10	S32						
<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.						
<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 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".						

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled\(\);
integer op2;
constant boolean round = TRUE;
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = SInt(Elem[D[n+r],e,esize]);
    constant op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    constant integer rdmlah = RShr(op3 + 2*(op1*op2), esize, round);
    constant (result, sat) = SignedSatQ(rdmlah, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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  
(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			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant add = FALSE; constant scalar_form = FALSE; constant esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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																															

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;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant add = FALSE; constant scalar_form = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = if size == '01' then UInt(M:Vm<3>) else UInt(M);
```

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				

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;
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
constant add = FALSE; constant scalar_form = FALSE; constant esize = 8 << UInt(size);
constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

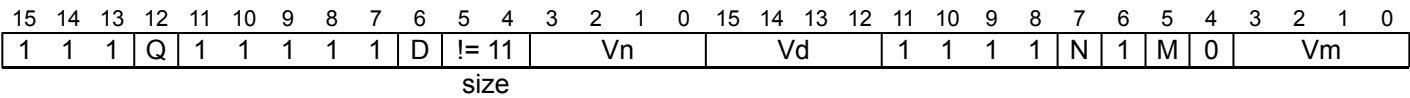
CONSTRAINED UNPREDICTABLE behavior

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)



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;
if InITBlock() then UNPREDICTABLE;
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant add = FALSE; constant scalar_form = TRUE;
constant d = UInt(D:Vd); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = if size == '01' then UInt(M:Vm<3>) else UInt(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.

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”: <table><tr><th>size</th><th>&lt;dt&gt;</th></tr><tr><td>01</td><td>S16</td></tr><tr><td>10</td><td>S32</td></tr></table>	size	<dt>	01	S16	10	S32
size	<dt>						
01	S16						
10	S32						
<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.						
<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 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".						

<Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

## Operation

```
EncodingSpecificOperations(); CheckAdvSIMDEnabled\(\);
integer op2;
constant boolean round = TRUE;
if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = SInt(Elem[D[n+r],e,esize]);
    constant op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    constant integer rdmlsh = RShr(op3 - 2*(op1*op2), esize, round);
    constant (result, sat) = SignedSatQ(rdmlsh, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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		

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;
if size == '00' || size == '11' then UNDEFINED;
constant scalar_form = FALSE;  constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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																															

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";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer      m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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	1	1	1	1	1	0	D	size	Vn				Vd				1	0	1	1	N	Q	M	0	Vm				

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;
if size == '00' || size == '11' then UNDEFINED;
constant scalar_form = FALSE;  constant esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
constant integer index = integer UNKNOWN;
```

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	Q	1	1	1	1	1	D	!= 11	Vn					Vd					1	1	0	1	N	1	M	0	Vm				
size																																	

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";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant scalar_form = TRUE;  constant d = UInt(D:Vd);  constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant integer m = if size == '01' then UInt(Vm<2:0>) else UInt(Vm);
constant integer index = 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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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.
- <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.
- <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[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".
- <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();
    integer op2;
    constant boolean round = TRUE;
    if scalar_form then op2 = SInt(Elem[D[m],index,esize]);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = SInt(Elem[D[n+r],e,esize]);
            if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
            constant integer rdmulh = RShr(2*op1*op2, esize, round);
            constant (result, sat) = SignedSatQ(rdmulh, esize);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = 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				

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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.



## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    bits(esize) result;
    boolean sat;
    for r = 0 to regs-1
        for e = 0 to elements-1
            integer element = Int(Elem[D[m+r], e, esize], unsigned);
            integer shift = SInt(Elem[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;
            (result, sat) = SatQ(element, esize, unsigned);
            Elem[D[d+r], e, esize] = result;
            if sat then FPSCR.QC = '1';
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			
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		
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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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	imm6						Vd				1	0	0	op	0	1	M	1	Vm			

#### Encoding for the Signed result variant

Applies when  $(!(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 1)$

`VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>`

#### Encoding for the Unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 0)$

`VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>`

#### Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize << 1) - UInt(imm6);
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = 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	0	op	0	1	M	1	Vm			

#### Encoding for the Signed result variant

Applies when  $(!(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 1)$

`VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>`

#### Encoding for the Unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 0)$

VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

## Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize << 1) - UInt(imm6);
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: see [Standard assembler syntax fields](#).

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

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	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();
    constant boolean round = TRUE;
    for e = 0 to elements-1
        constant operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        constant integer rshr = RShr(operand, shift_amount, round);
        constant (result, sat) = SatQ(rshr, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
        if sat then FPSCR.QC = '1';
```

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		
																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		
																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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = 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				



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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant shift = SInt(Elem[D[n+r],e,esize]<7:0>);
      constant operand = Int(Elem[D[m+r],e,esize], unsigned);
      boolean sat;
      bits(esize) result;
      if shift >= 0 then
        (result,sat) = SatQ(operand << shift, esize, unsigned);
      else
        (result,sat) = SatQ(operand >> -shift, esize, unsigned);
      Elem[D[d+r],e,esize] = result;
      if sat then FPSCR.QC = '1';
```



## 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 [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	1	1	op	L	Q	M	1	Vm			

#### Encoding for the 64-bit SIMD vector, signed result variant

Applies when  $(!(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ \text{op} == 1 \ \&\& \ Q == 0)$

VQSHL{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

#### Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ \text{op} == 0 \ \&\& \ Q == 0)$

VQSHLU{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

#### Encoding for the 128-bit SIMD vector, signed result variant

Applies when  $(!(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ \text{op} == 1 \ \&\& \ Q == 1)$

VQSHL{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

#### Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ \text{op} == 0 \ \&\& \ Q == 1)$

VQSHLU{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

#### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>;
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(L:imm6) - esize;
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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  $(!(imm6 == 000xxx \ \&\& \ L == 0) \ \&\& \ op == 1 \ \&\& \ Q == 0)$

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 64-bit SIMD vector, unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(imm6 == 000xxx \ \&\& \ L == 0) \ \&\& \ op == 0 \ \&\& \ Q == 0)$

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector, signed result variant

Applies when  $(!(imm6 == 000xxx \ \&\& \ L == 0) \ \&\& \ op == 1 \ \&\& \ Q == 1)$

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

Encoding for the 128-bit SIMD vector, unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(imm6 == 000xxx \ \&\& \ L == 0) \ \&\& \ op == 0 \ \&\& \ Q == 1)$

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(L:imm6) - esize;
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

<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.

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant operand = Int(Elem[D[m+r],e,esize], src_unsigned);
            constant (result, sat) = SatQ(operand << shift_amount, esize, dest_unsigned);
            Elem[D[d+r],e,esize] = result;
            if sat then FPSCR.QC = '1';

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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			
																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			
																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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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<0>:size”:

op<0>	size	<dt>
0	00	S16
0	01	S32
0	10	S64
0	11	RESERVED
1	00	U16
1	01	U32
1	10	U64
1	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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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	imm6						Vd				1	0	0	op	0	0	M	1	Vm			

#### Encoding for the Signed result variant

Applies when  $(!(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 1)$

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

#### Encoding for the Unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 0)$

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

#### Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (2 * esize) - UInt(imm6);
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = 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	0	op	0	0	M	1	Vm			

#### Encoding for the Signed result variant

Applies when  $(!(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 1)$

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

#### Encoding for the Unsigned result variant

Applies when  $(U == 1 \ \&\& \ !(\text{imm6} == 000\text{xxx}) \ \&\& \ \text{op} == 0)$



VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

## Decode for all variants of this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (2 * esize) - UInt(imm6);
constant src_unsigned = (U == '1' && op == '1'); constant dest_unsigned = (U == '1');
constant d = UInt(D:Vd); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

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

<type> For the signed result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
0	S
1	U

For the unsigned result variant: is the data type for the elements of the vectors, encoded in “U”:

U	<type>
1	S

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	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
        constant operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
        constant (result, sat) = SatQ(operand >> shift_amount, esize, dest_unsigned);
        Elem[D[d],e,esize] = result;
    if sat then FPSCR.QC = '1';
```

# 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					
																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			
																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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: 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.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
- <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.
- <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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant diff = (Int(Elem[D[n+r],e,esize], unsigned) -
                      Int(Elem[D[m+r],e,esize], unsigned));
      boolean sat;
      (Elem[D[d+r],e,esize], sat) = SatQ(diff, esize, unsigned);
      if sat then FPSCR.QC = '1';
```

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

31	30	29	28	27	26	25	24	23	22	21	20	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	!= 11	Vn			Vd			0 1 0 0			N	0	M	0	Vm							
										size																					

Encoding for the A1 variant

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = 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	!= 11	Vn			Vd			0	1	0	0	N	0	M	0	Vm						
size																															

Encoding for the T1 variant

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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();
    constant boolean round = TRUE;
    for e = 0 to elements-1
        constant result = (RShr(UInt(Elem[Qin[n>>1],e,2*esize] +
                                Elem[Qin[m>>1],e,2*esize]), esize, round));
        Elem[D[d],e,esize] = result<esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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					

### 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;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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		



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;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	10	U32
1	01	F16
1	10	F32

<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.

<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.

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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRecipEstimate(Elem[D[m+r],e,esize], fpcr);
            else
                Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *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	1	Vm			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.

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
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPREcipStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```



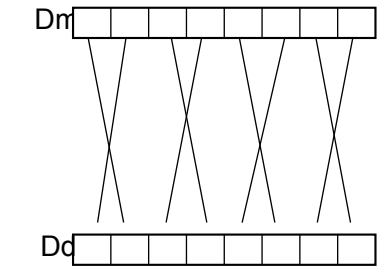
VREV16

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			
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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		
																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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	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.
- <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(); CheckAdvSIMDEnabled\(\);

bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = (element + elements_per_container) - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

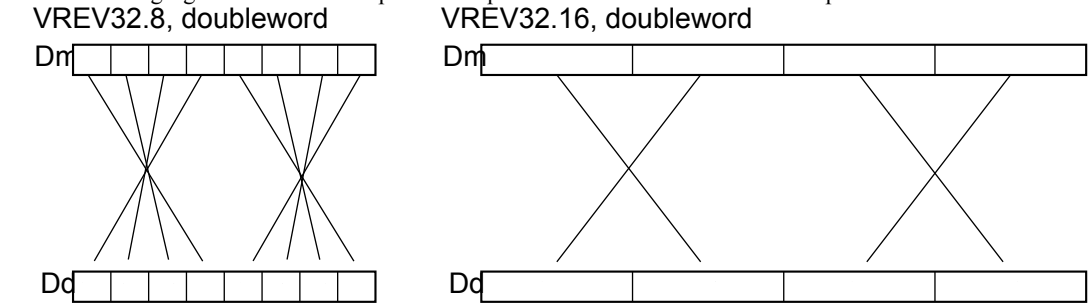


VREV32

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.



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				
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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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		
																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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	8
01	16
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.
- <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(); CheckAdvSIMDEnabled\(\);

bits(64) result;
integer element;
integer rev_element;
for r = 0 to regs-1
    element = 0;
    for c = 0 to containers-1
        rev_element = (element + elements_per_container) - 1;
        for e = 0 to elements_per_container-1
            Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
            element = element + 1;
            rev_element = rev_element - 1;
        D[d+r] = result;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

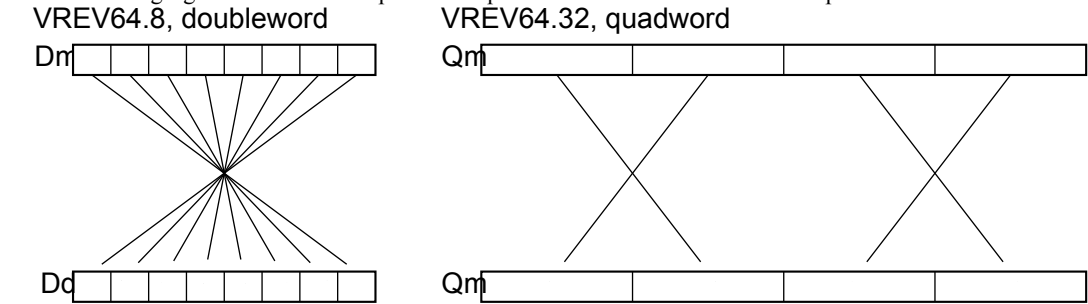
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				
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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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	0	Q	M	0		Vm		
																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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

constant esize = 8 << UInt(size);
constant integer container_size = 64 >> UInt(op);
constant integer containers = 64 DIV container_size;
constant integer elements_per_container = container_size DIV esize;

constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	8
01	16
10	32
11	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.
- <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(); CheckAdvSIMDEnabled\(\);

    bits(64) result;
    integer element;
    integer rev_element;
    for r = 0 to regs-1
        element = 0;
        for c = 0 to containers-1
            rev_element = (element + elements_per_container) - 1;
            for e = 0 to elements_per_container-1
                Elem[result, rev_element, esize] = Elem[D[m+r], element, esize];
                element = element + 1;
                rev_element = rev_element - 1;
            D[d+r] = result;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.

- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				

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;
if size == '11' then UNDEFINED;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '11' then UNDEFINED;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant op1 = Int(Elem[D[n+r],e,esize], unsigned);
            constant op2 = Int(Elem[D[m+r],e,esize], unsigned);
            constant result = (op1 + op2 + 1) >> 1;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.



- The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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				
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";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				
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";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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
- <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.
- <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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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	!= 00		0	1	M	0	Vm			
RM																size															

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;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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	0	Vd				1	0	!= 00		0	1	M	0	Vm			
RM																size															

**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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <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.
<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);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```

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				
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";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				
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";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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
- <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.
- <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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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	!= 00	0	1	M	0	Vm				
RM																size															

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;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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	!= 00	0	1	M	0	Vm				
RM																size															



**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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <a href="#">Standard assembler syntax fields</a> .
<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);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```

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				
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";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				
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";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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  |
- <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.
- <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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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	!= 00	0	1	M	0	Vm				
RM																size															

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;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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	!= 00	0	1	M	0	Vm				
RM																size															

**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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <a href="#">Standard assembler syntax fields</a> .
<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);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```

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					
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";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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					
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";
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
constant rounding = FPDecodeRM(op<2>:NOT(op<1>)); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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
- <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.
- <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(); CheckAdvSIMDEnabled();
constant FPCR_Type fpcr = StandardFPCR();
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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	!= 00	0	1	M	0	Vm				
RM																size															

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;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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	!= 00	0	1	M	0	Vm				
RM																size															



**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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant rounding = FPDecodeRM(RM); constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <a href="#">Standard assembler syntax fields</a> .
<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);
constant FPCR_Type fpcr = EffectiveFPCR();
case esize of
  when 16
    H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```

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																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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant boolean zero_rounding = op == '1';
constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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				
																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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant boolean zero_rounding = op == '1';
constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <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.
<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);
    constant FPCR_Type fpcr = EffectiveFPCR();
    constant rounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16
            H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```



## 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			

#### 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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPRounding\_TIEEVEN; constant exact = TRUE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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			

#### 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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPRounding\_TIEEVEN; constant exact = TRUE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

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 “size”:

size	<dt>
01	F16
10	F32

<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.

<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(); CheckAdvSIMDEnabled\(\);
constant FPCR\_Type fpcr = StandardFPCR\(\);
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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																															

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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant exact = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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				

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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant exact = TRUE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <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.
<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);
  constant FPCR_Type fpcr = EffectiveFPCR();
  constant rounding = FPRoundingMode(fpcr);
  case esize of
    when 16
      H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```





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				

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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPRounding\_ZERO; constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				

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;
if (size == '01' && !IsFeatureImplemented(FEAT_FP16)) || size IN {'00', '11'} then UNDEFINED;
constant rounding = FPRounding\_ZERO; constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
if InITBlock() then UNPREDICTABLE;
```

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 “size”:

size	<dt>
01	F16
10	F32

<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.

<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(); CheckAdvSIMDEnabled\(\);
constant FPCR\_Type fpcr = StandardFPCR\(\);
for r = 0 to regs-1
  for e = 0 to elements-1
    constant op1 = Elem[D[m+r],e,esize];
    constant result = FPRoundInt(op1, fpcr, rounding, exact);
    Elem[D[d+r],e,esize] = result;
```

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												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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant boolean zero_rounding = op == '1';
constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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				
																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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant boolean zero_rounding = op == '1';
constant exact = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <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.
<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);
    constant FPCR_Type fpcr = EffectiveFPCR();
    constant rounding = if zero_rounding then FPRounding_ZERO else FPRoundingMode(fpcr);
    case esize of
        when 16
            H[d] = FPRoundInt(H[m], fpcr, rounding, exact);
        when 32
            S[d] = FPRoundInt(S[m], fpcr, rounding, exact);
        when 64
            D[d] = FPRoundInt(D[m], fpcr, rounding, exact);
```



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			

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size);  constant elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant n = UInt(N:Vn);
constant regs = 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				

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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

- <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.
- <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.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  integer result;
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant integer element = Int(Elem[D[m+r], e, esize], unsigned);
      integer shift = SInt(Elem[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;
      Elem[D[d+r], e, esize] = result<esize-1:0>;
```



## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>}, {<Dm>}, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>}, {<Qm>}, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>,<Dm>,<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>,<Qm>,<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>;
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

<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>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant boolean round = TRUE;
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant result = RShr(Int(Elem[D[m+r],e,esize], unsigned), shift_amount, round);
            Elem[D[d+r],e,esize] = result<size-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	0	1	N	Q	M	1	Vm			

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			

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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.

Operation

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

# 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

31	30	29	28	27	26	25	24	23	22	21	20	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				1	0	0	0	0	1	M	1	Vm			

### Encoding for the A1 variant

Applies when (imm6 != 000xxx)

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

### Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize << 1) - UInt(imm6);
constant d = UInt(D:Vd); constant m = 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	imm6						Vd				1	0	0	0	0	1	M	1	Vm			

### Encoding for the T1 variant

Applies when (imm6 != 000xxx)

VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>

### Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize << 1) - UInt(imm6);
constant d = UInt(D:Vd); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

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

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	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();
    constant boolean round = TRUE;
    for e = 0 to elements-1
        constant result = RShr(UInt(Elem[Qin[m>>1],e,2*esize]), shift_amount, round);
        Elem[D[d],e,esize] = result<size-1:0>;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



# 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	M	0	Vm					

### 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			

### 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

<q> For encoding T1: 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.



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 *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	1	Q	M	0	Vm					

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;
if (size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0')) || size IN {'00', '11'} then
    UNDEFINED;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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				

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;
if size == '01' && (!IsFeatureImplemented(FEAT_FP16) || F == '0') then UNDEFINED;
if size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant floating_point = (F == '1');
constant integer esize = 8 << UInt(size);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	10	U32
1	01	F16
1	10	F32

<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.

<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.

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();
    constant FPCR\_Type fpcr = StandardFPCR();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if floating_point then
                Elem[D[d+r],e,esize] = FPRSqrtEstimate(Elem[D[m+r],e,esize], fpcr);
            else
                Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *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	1	Vm					

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
constant regs = 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					

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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”:

sz	<dt>
0	F32
1	F16
- <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.
- <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.

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
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = FPRSqrtStep(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize]);
```





## 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 [VSR4](#).

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			

#### Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>}, {<Dm>}, #<imm>
```

#### Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>}, {<Qm>}, #<imm>
```

#### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>,<Dm>,<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>,<Qm>,<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>;
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

<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>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant boolean round = TRUE;
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant result = RShr(Int(Elem[D[m+r],e,esize], unsigned), shift_amount, round);
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

31	30	29	28	27	26	25	24	23	22	21	20	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	!= 11	Vn			Vd			0 1 1 0			N	0	M	0	Vm							
size																															

Encoding for the A1 variant

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	!= 11	Vn				Vd				0	1	1	0	N	0	M	0	Vm				
size																															

Encoding for the T1 variant

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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();
    constant boolean round = TRUE;
    for e = 0 to elements-1
        constant result = (RShr(UInt(Elem[Qin[n>>1],e,2*esize] -
                                Elem[Qin[m>>1],e,2*esize]), esize, round));
        Elem[D[d],e,esize] = result<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			
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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean signed = (U=='0');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<3:0>);
constant integer index = UInt(M);
constant integer esize = 32;
constant integer regs = 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					
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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_DotProd) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean signed = (U=='0');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<3:0>);
constant integer index = UInt(M);
constant integer esize = 32;
constant integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

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

Operation

```
bits(64) operand1;
constant bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

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					
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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant boolean signed = U==0';
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer esize = 32;
constant integer regs = 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					
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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_DotProd) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant boolean signed = U=='0';
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer esize = 32;
constant integer regs = if Q == '1' then 2 else 1;
```

## 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.
<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

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	!= 00		N	0	M	0	Vm			
size																															

### 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

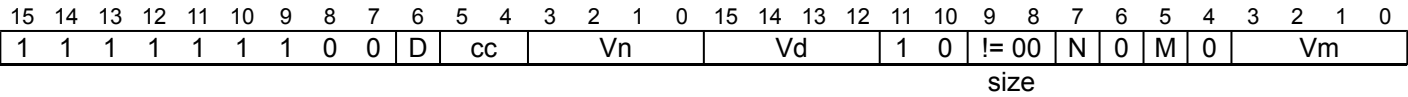
Applies when (cc == 01 && size == 11)

```
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;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant bits(4) condition = cc:(cc<1> EOR cc<0>):'0';
```

T1



### 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 UNPREDICTABLE;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant bits(4) condition = cc:(cc<1> EOR 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

<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.
<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(); 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];
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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	1	D	imm6						Vd				0	1	0	1	L	Q	M	1	Vm			

#### Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

`VSHL{<c>}{<q>}.I<size> {<Dd>}, <Dm>, #<imm>`

#### Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

`VSHL{<c>}{<q>}.I<size> {<Qd>}, <Qm>, #<imm>`

#### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(L:imm6) - esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VSHL{<c>}{<q>}.I<size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VSHL{<c>}{<q>}.I<size> {<Qd>}, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(L:imm6) - esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.  
<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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = LSL(Elem[D[m+r],e,esize], shift_amount);
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.





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				

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = 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				

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;
constant unsigned = (U == '1');
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant n = UInt(N:Vn);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
- <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.
- <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.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  for r = 0 to regs-1
    for e = 0 to elements-1
      constant shift = SInt(Elem[D[n+r],e,esize]<7:0>);
      integer result;
      if shift >= 0 then
        result = Int(Elem[D[m+r],e,esize], unsigned) << shift;
      else
        result = Int(Elem[D[m+r],e,esize], unsigned) >> -shift;
      Elem[D[d+r],e,esize] = result<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

### Encoding for the A1 variant

Applies when (imm6 != 000xxx)

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

### Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 IN {'001000', '010000', '100000'} then SEE "VMOVL";
if Vd<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(imm6) - esize;
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = 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				

### Encoding for the A2 variant

```
VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>
```

### Decode for this encoding

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant shift_amount = esize;
constant unsigned = FALSE; // Or TRUE without change of functionality
constant d = UInt(D:Vd); constant m = 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			

### Encoding for the T1 variant

Applies when (imm6 != 000xxx)

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

## Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if imm6 IN {'001000', '010000', '100000'} then SEE "VMOVL";
if Vd<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(imm6) - esize;
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = 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				

## Encoding for the T2 variant

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

## Decode for this encoding

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant shift_amount = esize;
constant unsigned = FALSE; // Or TRUE without change of functionality
constant d = UInt(D:Vd); constant m = 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 encoding A1 and A2: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1 and T2: see [Standard assembler syntax fields](#).

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

<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.

<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>.

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for e = 0 to elements-1
        constant result = Int(Elem[Din[m],e,esize], unsigned) << shift_amount;
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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			

#### Encoding for the 64-bit SIMD vector variant

Applies when  $(!(\text{imm6} == 000\text{xxx} \ \&\& \ L == 0) \ \&\& \ Q == 0)$

```
VSHR{<c>}{<q>}.<type><size> {<Dd>}, {<Dm>}, #<imm>
```

#### Encoding for the 128-bit SIMD vector variant

Applies when  $(!(\text{imm6} == 000\text{xxx} \ \&\& \ L == 0) \ \&\& \ Q == 1)$

```
VSHR{<c>}{<q>}.<type><size> {<Qd>}, {<Qm>}, #<imm>
```

#### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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			



Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt (L:imm6);
constant unsigned = (U == '1'); constant d = UInt (D:Vd); constant m = UInt (M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

<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>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant result = Int (Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = result<esize-1:0>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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	0	1	N	Q	M	1	Vm			

### 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			

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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.
- <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.
- <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.

Operation

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

# 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

31	30	29	28	27	26	25	24	23	22	21	20	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				1	0	0	0	0	0	0	M	1	Vm		

### Encoding for the A1 variant

Applies when (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

### Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (2 * esize) - UInt(imm6);
constant d = UInt(D:Vd); constant m = 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	imm6						Vd				1	0	0	0	0	0	0	M	1	Vm		

### Encoding for the T1 variant

Applies when (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

### Decode for this encoding

```
if imm6 == '000xxx' then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << HighestSetBit(imm6<5:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (2 * esize) - UInt(imm6);
constant d = UInt(D:Vd); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1: see [Standard assembler syntax fields](#).

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

<size> Is the data size for the elements of the vectors, encoded in “imm6<5:3>”:

imm6<5:3>	<size>
001	16
01x	32
1xx	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
        constant result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<size-1:0>;
```

# 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					

### 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			

### 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.

<q> For encoding T1: 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.





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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

VSLI{<c>}{<q>}.<size> {<Dd>,,} <Dm>,, #<imm>

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

VSLI{<c>}{<q>}.<size> {<Qd>,,} <Qm>,, #<imm>

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt(L:imm6) - esize;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VSLI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VSLI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ (L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = UInt (L:imm6) - esize;
constant d = UInt (D:Vd); constant m = UInt (M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64
- <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.
- <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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant shifted_op = LSL(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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)

31	30	29	28	27	26	25	24	23	22	21	20	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	0	N	1	M	0	Vm					
B												U																			

### Encoding for the A1 variant

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

### Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
```

### 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	0	D	1	0	Vn			Vd			1	1	0	0	N	1	M	0	Vm					
B												U																			

### Encoding for the T1 variant

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

### Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = 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();
constant bits(128) operand1 = Q[n>>1];
constant bits(128) operand2 = Q[m>>1];
constant bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

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 *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	0	1	Vd				1	0	size		1	1	M	0	Vm			
cond																															

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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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			

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;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer m = 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 <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.
<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);
    constant FPCR_Type fpcr = EffectiveFPCR();
    case esize of
        when 16 H[d] = FPSqrt(H[m], fpcr);
        when 32 S[d] = FPSqrt(S[m], fpcr);
        when 64 D[d] = FPSqrt(D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 [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	1	L	Q	M	1	Vm			

#### Encoding for the 64-bit SIMD vector variant

Applies when  $(!(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ Q == 0)$

```
VSRA{<c>}{<q>}.<type><size> {<Dd>}, {<Dm>}, #<imm>
```

#### Encoding for the 128-bit SIMD vector variant

Applies when  $(!(\text{imm6} == 000xxx \ \&\& \ L == 0) \ \&\& \ Q == 1)$

```
VSRA{<c>}{<q>}.<type><size> {<Qd>}, {<Qm>}, #<imm>
```

#### Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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			



## Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VSRA{<c>}{<q>}.<type><size> {<Dd>,} <Dm>, #<imm>
```

## Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VSRA{<c>}{<q>}.<type><size> {<Qd>, } <Qm>, #<imm>
```

## Decode for all variants of this encoding

```

if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant unsigned = (U == '1'); constant d = UInt(D:Vd); constant m = UInt(M:Vm);
constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.

For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64

<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.

**<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>.

## Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();    CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant result = Int(Elem[D[m+r],e,esize], unsigned) >> shift_amount;
            Elem[D[d+r],e,esize] = Elem[D[d+r],e,esize] + result;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

`VSRI{<c>}{<q>}.<size> {<Dd>,,} <Dm>, #<imm>`

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

`VSRI{<c>}{<q>}.<size> {<Qd>,,} <Qm>, #<imm>`

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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			

Encoding for the 64-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 0)`

```
VSRI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

Encoding for the 128-bit SIMD vector variant

Applies when `(!(imm6 == 000xxx && L == 0) && Q == 1)`

```
VSRI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

Decode for all variants of this encoding

```
if (L:imm6) == '0000xxx' then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer esize = 8 << HighestSetBitNZ((L:imm6)<6:3>);
constant integer elements = 64 DIV esize;
constant integer shift_amount = (esize * 2) - UInt(L:imm6);
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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<5:3>”:

L	imm6<5:3>	<size>
0	001	8
0	01x	16
0	1xx	32
1	xxx	64
- <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.
- <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>.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled();
    constant mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            constant shifted_op = LSR(Elem[D[m+r],e,esize], shift_amount);
            Elem[D[d+r],e,esize] = (Elem[D[d+r],e,esize] AND NOT(mask)) OR shifted_op;
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:

- The values of the data supplied in any of its registers.
- The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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					

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

```
constant regs = 1; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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					

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

```
constant regs = 2; if align == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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	0	Rn				Vd				0	1	1	0	size		align		Rm			

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

```
constant regs = 3; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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			



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

```
constant regs = 4;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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					

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
constant regs = 1; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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							

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]}!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
constant regs = 2; if align == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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					

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

```
constant regs = 3; if align<1> == '1' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	0	size		align		Rm			

Encoding for the Offset variant

Applies when (Rm == 1111)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]]!
```

Encoding for the Post-indexed variant

Applies when (Rm != 11x1)

```
VSTl{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
constant regs = 4;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size);  constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd);  constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d+regs > 32 then UNPREDICTABLE;
```

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 *VSTl (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 encoding A1, A2, A3 and A4: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2, T3 and T4: 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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address, ebytes] = Elem[D[d+r], e, 8*ebytes];
            else
                if !IsAligned(address, ebytes) && AlignmentEnforced() then
                    constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
                    AArch32.Abort(fault);

                constant bits(64) data = Elem[D[d+r], e, 64];
                if BigEndian(AccessType_ASIMD) then
                    MemU[address, 4] = data<63:32>;
                    MemU[address+4, 4] = data<31:0>;
                else
                    MemU[address, 4] = data<31:0>;
                    MemU[address+4, 4] = data<63:32>;

                address = address + ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 8*regs;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 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			
size																															

#### 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;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant alignment = 1;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

### 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			
																size															

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;
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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			
size																															

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)

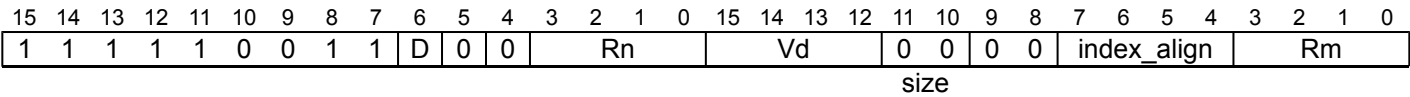
```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant alignment = if index_align<1:0> == '00' then 1 else 4;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```



T1



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

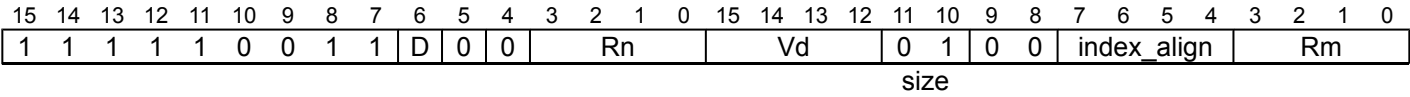
Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant alignment = 1;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T2



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

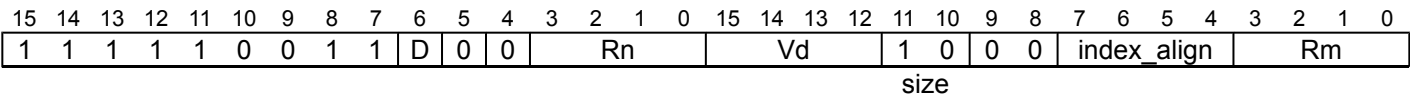
Applies when (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

T3



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;
if index_align<2> != '0' then UNDEFINED;
if index_align<1:0> != '00' && index_align<1:0> != '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant alignment = if index_align<1:0> == '00' then 1 else 4;
constant d = UInt(D:Vd); constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;
```

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

Assembler Symbols

<c> For encoding A1, A2 and A3: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1, T2 and T3: 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    MemU[address,ebytes] = Elem[D[d],index,8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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							
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

```
constant pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
constant inc = if itype == '1001' then 2 else 1;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

### 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					

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

```
constant pairs = 2;  constant inc = 2;
if size == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size);  constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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					
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

```
constant pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
constant inc = if itype == '1001' then 2 else 1;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	1	size		align		Rm					

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

```
constant pairs = 2;  constant inc = 2;
if size == '11' then UNDEFINED;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size);  constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd);  constant d2 = d + inc;
constant n = UInt(Rn);  constant m = UInt(Rm);
constant wback = (m != 15);  constant register_index = (m != 15 && m != 13);
if n == 15 || d2+pairs > 32 then UNPREDICTABLE;
```

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 encoding A1 and A2: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1 and T2: 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	RESERVED

<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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for r = 0 to pairs-1
        for e = 0 to elements-1
            MemU[address, ebytes] = Elem[D[d+r], e, 8*ebytes];
            MemU[address+ebytes, ebytes] = Elem[D[d2+r], e, 8*ebytes];
            address = address + 2*ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 16*pairs;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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						
size																															

#### 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;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

### 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			
size																															

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;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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						
size																															

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

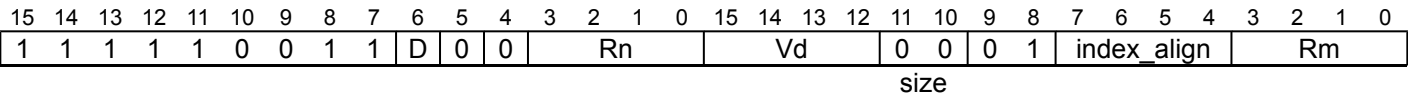
```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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



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

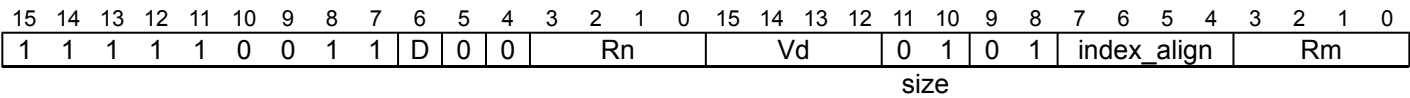
```
if size == '11' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	1	index_align				Rm					
size																															

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;
if index_align<1> != '0' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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 encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2 and T3: 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 two SIMD&FP registers holding the element.  
The list must be one of:  
{ <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.  
  
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 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.

: 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    MemU[address, ebytes] = Elem[D[d], index, 8*ebytes];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index, 8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 2*ebytes;
```



**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 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 [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			1	0	x	size		align		Rm			
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

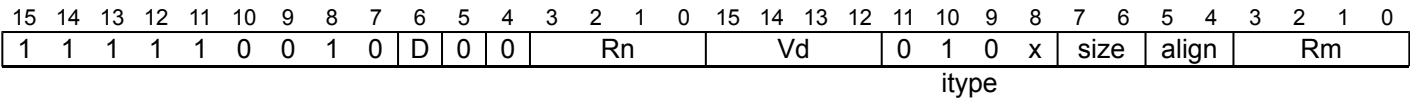
```
if size == '11' || align<1> == '1' then UNDEFINED;
integer inc;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
constant alignment = if align<0> == '0' then 1 else 8;
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

## 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



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;
integer inc;
case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
constant alignment = if align<0> == '0' then 1 else 8;
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.
- For encoding T1: 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	RESERVED

<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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
                                                            tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for e = 0 to elements-1
        MemU[address, ebytes] = Elem[D[d], e, 8*ebytes];
        MemU[address+ebytes, ebytes] = Elem[D[d2], e, 8*ebytes];
        MemU[address+2*ebytes, ebytes] = Elem[D[d3], e, 8*ebytes];
        address = address + 3*ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 24;
```

Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

## 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						
size																															

#### 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;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

### 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						
size																															

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

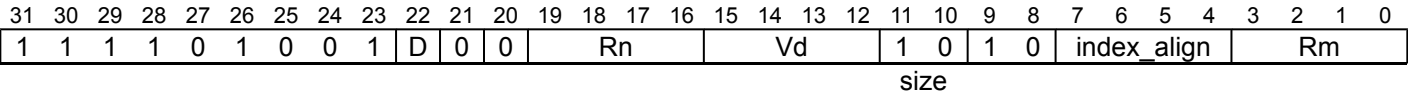
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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



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

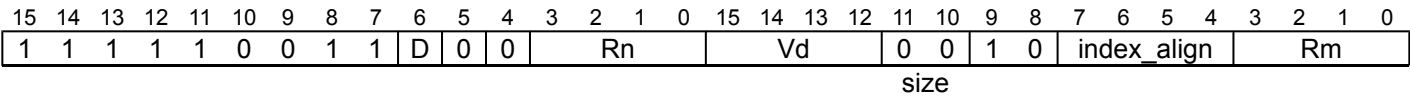
```
if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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



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;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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	0	index_align			Rm						
size																															



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

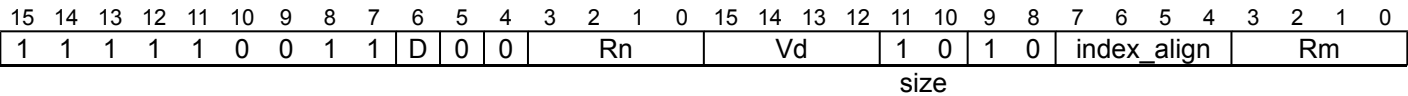
```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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;
if index_align<1:0> != '00' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;
```

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> For encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2 and T3: 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 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();
    constant address = R[n];
    MemU[address, ebytes] = Elem[D[d], index, 8*ebytes];
    MemU[address+ebytes, ebytes] = Elem[D[d2], index, 8*ebytes];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index, 8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 3*ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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							
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;
if itype != '000x' then SEE "Related encodings";
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### 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					
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;
if itype != '000x' then SEE "Related encodings";
constant inc = if itype<0> == '0' then 1 else 2;
constant alignment = if align == '00' then 1 else 4 << UInt(align);
constant ebytes = 1 << UInt(size); constant elements = 8 DIV ebytes;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	RESERVED

- <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();

    address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp_STORE, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    for e = 0 to elements-1
        MemU[address, ebytes] = Elem[D[d], e, 8*ebytes];
        MemU[address+ebytes, ebytes] = Elem[D[d2], e, 8*ebytes];
        MemU[address+2*ebytes, ebytes] = Elem[D[d3], e, 8*ebytes];
        MemU[address+3*ebytes, ebytes] = Elem[D[d4], e, 8*ebytes];
        address = address + 4*ebytes;

    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 32;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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				size			

#### 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;
if size != '00' then SEE "Related encodings";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

### 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				size			

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;
if size != '01' then SEE "Related encodings";
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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						
size																															

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;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

## 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			
size																															

## 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;
if size != '00' then SEE "Related encodings";
constant ebytes = 1; constant index = UInt(index_align<3:1>); constant inc = 1;
constant alignment = if index_align<0> == '0' then 1 else 4;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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						
size																															

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;
if size != '01' then SEE "Related encodings";
constant ebytes = 2; constant index = UInt(index_align<3:2>);
constant inc = if index_align<1> == '0' then 1 else 2;
constant alignment = if index_align<0> == '0' then 1 else 8;
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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						
size																															

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 != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

Encoding for the Post-indexed variant

Applies when (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Decode for all variants of this encoding

```
if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
constant ebytes = 4; constant index = UInt(index_align<3>);
constant inc = if index_align<2> == '0' then 1 else 2;
constant alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
constant d = UInt(D:Vd); constant d2 = d + inc;
constant d3 = d2 + inc; constant d4 = d3 + inc;
constant n = UInt(Rn); constant m = UInt(Rm);
constant wback = (m != 15); constant register_index = (m != 15 && m != 13);
if n == 15 || d4 > 31 then UNPREDICTABLE;
```

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 encoding A1, A2 and A3: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1, T2 and T3: 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();

    constant address = R[n];

    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescASIMD(MemOp\_STORE, nontemporal,
        tagchecked, privileged);

    if !IsAligned(address, alignment) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    MemU[address,          ebytes] = Elem[D[d], index, 8*ebytes];
    MemU[address+ebytes,    ebytes] = Elem[D[d2], index, 8*ebytes];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index, 8*ebytes];
    MemU[address+3*ebytes, ebytes] = Elem[D[d4], index, 8*ebytes];
    if wback then
        if register_index then
            R[n] = R[n] + R[m];
        else
            R[n] = R[n] + 4*ebytes;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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	imm8<7:1>						0	
cond																imm8<0>															

#### 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>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

#### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

#### 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																															

### 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>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

### Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = TRUE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(Vd:D); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32); constant regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```

### 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	imm8<7:1>				0						
																															imm8<0>			

### 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>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

## Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = FALSE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(D:Vd); constant n = UInt(Rn); constant imm32 = ZeroExtend(imm8:'00', 32);
constant regs = UInt(imm8) DIV 2; // If UInt(imm8) is odd, see "FSTDDBMX, FSTMIAX".
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || regs > 16 || (d+regs) > 32 then UNPREDICTABLE;
if imm8<0> == '1' && (d+regs) > 16 then UNPREDICTABLE;
```

## 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

15	14	13	12	11	10	9	8	7	6	5	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	0	imm8							

## 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>
```

```
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

## Decode for all variants of this encoding

```
if P == '0' && U == '0' && W == '0' then SEE "Related encodings";
if P == '1' && W == '0' then SEE "VSTR";
if P == U && W == '1' then UNDEFINED;
// Remaining combinations are PUW = 010 (IA without !), 011 (IA with !), 101 (DB with !)
constant single_regs = TRUE; constant add = (U == '1'); constant wback = (W == '1');
constant d = UInt(Vd:D); constant n = UInt(Rn);
constant imm32 = ZeroExtend(imm8:'00', 32); constant regs = UInt(imm8);
if n == 15 && (wback || CurrentInstrSet() != InstrSet_A32) then UNPREDICTABLE;
if regs == 0 || (d+regs) > 32 then UNPREDICTABLE;
```



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 <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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.
!	Specifies base register writeback. Encoded in the "W" field as 1 if present, otherwise 0.
<sgreglist>	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.

Alias Conditions

Alias	Is preferred when
<a href="#">VPUSH</a>	<code>P == '1' &amp;&amp; U == '0' &amp;&amp; W == '1' &amp;&amp; Rn == '1101'</code>

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = if add then R[n] else R[n]-imm32;
    for r = 0 to regs-1
        if single_regs then
            MemA[address,4] = 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[address,4] = D[d+r]<63:32>;
                MemA[address+4,4] = D[d+r]<31:0>;
            else
                MemA[address,4] = D[d+r]<31:0>;
                MemA[address+4,4] = D[d+r]<63:32>;

            address = address+8;

    if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

**Operational information**

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# 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																															

### 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;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

### 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							

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;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant esize = 8 << UInt(size); constant add = (U == '1');
constant imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant n = UInt(Rn);
if n == 15 && CurrentInstrSet() != InstrSet_A32 then UNPREDICTABLE;
```

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 <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
.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.						
.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.						
<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><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 single-precision scalar or double-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. For the half-precision scalar variant: 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.						

## Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    constant address = if add then (R[n] + imm32) else (R[n] - imm32);
    case esize of
        when 16
            MemA[address,2] = H[d];
        when 32
            MemA[address,4] = S[d];
        when 64
            // Store as two word-aligned words in the correct order for current endianness.
            if BigEndian(AccessType ASIMD) then
                MemA[address,4] = D[d]<63:32>;
                MemA[address+4,4] = D[d]<31:0>;
            else
                MemA[address,4] = D[d]<31:0>;
                MemA[address+4,4] = D[d]<63:32>;
```

## Operational information

If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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																															

### 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{&lt;c&gt;}{&lt;q&gt;}.F32 {&lt;Sd&gt;, } &lt;Sn&gt;, &lt;Sm&gt;

## Encoding for the Double-precision scalar variant

Applies when (size == 11)

VSUB{&lt;c&gt;}{&lt;q&gt;}.F64 {&lt;Dd&gt;,&lt;Dn&gt;,&lt;Dm&gt;

### Decode for all variants of this encoding

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;

```

## 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					

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;
if sz == '1' && !IsFeatureImplemented(FEAT_FP16) then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
constant advsimd = TRUE;
constant integer esize = 32 >> UInt(sz);
constant integer elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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			



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;
if size == '00' || (size == '01' && !IsFeatureImplemented(FEAT_FP16)) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
constant advsimd = FALSE;
constant integer esize = 8 << UInt(size);
constant integer d = if size == '11' then UInt(D:Vd) else UInt(Vd:D);
constant integer n = if size == '11' then UInt(N:Vn) else UInt(Vn:N);
constant integer m = if size == '11' then UInt(M:Vm) else UInt(Vm:M);
constant integer regs = integer UNKNOWN; constant integer elements = integer UNKNOWN;
```

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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding A2, T1 and T2: 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”:
- | sz | <dt> |
|----|------|
| 0  | F32  |
| 1  | F16  |
- <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.
- <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.
- <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
        constant FPCR\_Type fpcr = StandardFPCR();
        for r = 0 to regs-1
            for e = 0 to elements-1
                Elem[D[d+r],e,esize] = FPSub(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], fpcr);
    else // VFP instruction
        constant FPCR\_Type fpcr = EffectiveFPCR();
        case esize of
            when 16
                H[d] = FPSub(H[n], H[m], fpcr);
            when 32
                S[d] = FPSub(S[n], S[m], fpcr);
            when 64
                D[d] = FPSub(D[n], D[m], fpcr);
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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

#### 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;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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

#### 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;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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
11	I64

- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            Elem[D[d+r],e,esize] = Elem[D[n+r],e,esize] - Elem[D[m+r],e,esize];
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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

31	30	29	28	27	26	25	24	23	22	21	20	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 0			N	0	M	0	Vm							
size																															

Encoding for the A1 variant

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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	!= 11	Vn				Vd				0	1	1	0	N	0	M	0	Vm				
size																															

Encoding for the T1 variant

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = 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 encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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();
    for e = 0 to elements-1
        constant result = Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize];
        Elem[D[d],e,esize] = result<2*esize-1:esize>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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

31	30	29	28	27	26	25	24	23	22	21	20	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	0	M	0		Vm
size												op																			

Encoding for the A1 variant

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vsubw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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	1	0	N	0	M	0		Vm
size												op																			

Encoding for the T1 variant

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vsubw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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 encoding A1: see Standard assembler syntax fields. This encoding must be unconditional.  
For encoding T1: 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
        integer op1;
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        constant result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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 1			1	N	0	M	0	Vm			
size											op																					

Encoding for the A1 variant

VSUBW{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vsubw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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	1	1	N	0	M	0	Vm				
size											op																				

Encoding for the T1 variant

VSUBW{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>

Decode for this encoding

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
constant unsigned = (U == '1'); constant is_vsubw = (op == '1');
constant integer esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant elements = 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 encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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
        integer op1;
        if is_vsubw then
            op1 = Int(Elem[Qin[n>>1],e,2*esize], unsigned);
        else
            op1 = Int(Elem[Din[n],e,esize], unsigned);
        constant result = op1 - Int(Elem[Din[m],e,esize], unsigned);
        Elem[Q[d>>1],e,2*esize] = result<2*esize-1:0>;

```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

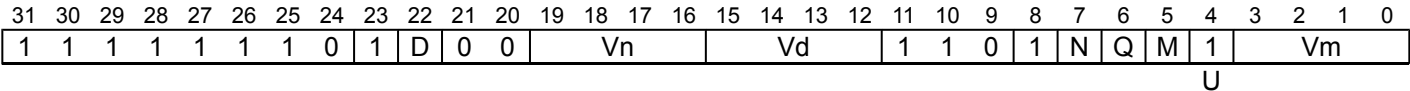
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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](#).18MM 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

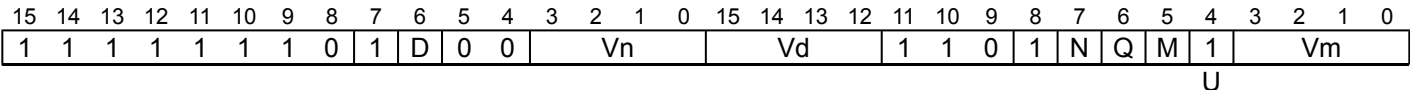
Applies when (Q == 1)

VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]

Decode for all variants of this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean op1_unsigned = (U == '0');
constant boolean op2_unsigned = (U == '1');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = 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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean op1_unsigned = (U == '0');
constant boolean op2_unsigned = (U == '1');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

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

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
  operand1 = Din[n+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(32) res = Elem[result, e, 32];
    for b = 0 to 3
      element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
      element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
      res = res + element1 * element2;
    Elem[result, e, 32] = res;
  D[d+r] = result;
```

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 *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	0	0	1	0	Vd				0	0	0	0	0	Q	M	0	Vm			
size																															

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = 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	Q	M	0	Vm			
size																															

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant d = UInt(D:Vd);  constant m = UInt(M:Vm);  constant regs = if Q == '0' then 1 else 2;
```

## Assembler Symbols

<c>	For encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<dt>	An optional data type. It is ignored by assemblers, and does not affect the encoding.
<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.
<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(); CheckAdvSIMDEnabled();
    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            D[d+r] = Din[m+r];
            D[m+r] = Din[d+r];
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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 *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	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

```
constant is_vtbl = (op == '0');  constant length = UInt(len)+1;
constant d = UInt(D:Vd);  constant n = UInt(N:Vn);  constant m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

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

```
constant is_vtbl = (op == '0'); constant length = UInt(len)+1;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
if n+length > 32 then UNPREDICTABLE;
```

## CONSTRAINED UNPREDICTABLE behavior

If  $n + \text{length} > 32$ , 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 encoding A1: see <a href="#">Standard assembler syntax fields</a> . This encoding must be unconditional. For encoding T1: see <a href="#">Standard assembler syntax fields</a> .
<q>	See <a href="#">Standard assembler syntax fields</a> .
<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: <div><b>{&lt;Dn&gt;}</b> Encoded as len = 0b00.</div> <div><b>{&lt;Dn&gt;, &lt;Dn+1&gt;}</b> Encoded as len = 0b01.</div> <div><b>{&lt;Dn&gt;, &lt;Dn+1&gt;, &lt;Dn+2&gt;}</b> Encoded as len = 0b10.</div> <div><b>{&lt;Dn&gt;, &lt;Dn+1&gt;, &lt;Dn+2&gt;, &lt;Dn+3&gt;}</b> Encoded as len = 0b11.</div>
<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.
    constant table3 = if length == 4 then D[n+3] else Zeros(64);
    constant table2 = if length >= 3 then D[n+2] else Zeros(64);
    constant table1 = if length >= 2 then D[n+1] else Zeros(64);
    constant table = table3 : table2 : table1 : D[n];

    for i = 0 to 7
        constant index = UInt(Elem[D[m],i,8]);
        if index < 8*length then
            Elem[D[d],i,8] = Elem[table,index,8];
        else
            if is_vtbl then
                Elem[D[d],i,8] = Zeros(8);
            // else Elem[D[d],i,8] unchanged
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

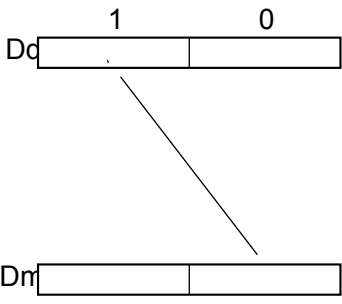
Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

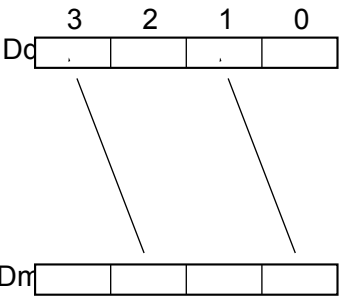
VTRN

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.

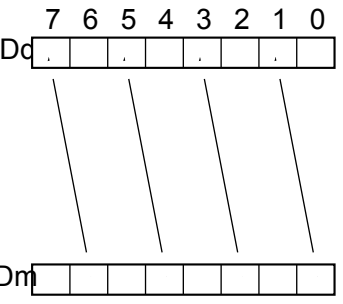
VTRN.32



VTRN.16



VTRN.8



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					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = 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				

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant m = UInt(M:Vm); constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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	8
01	16
10	32
11	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.
- <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(); CheckAdvSIMDEnabled();
    constant h = elements DIV 2;

    for r = 0 to regs-1
        if d == m then
            D[d+r] = bits(64) UNKNOWN;
        else
            for e = 0 to h-1
                Elem[D[d+r], 2*e+1, esize] = Elem[Din[m+r], 2*e, esize];
                Elem[D[m+r], 2*e, esize] = Elem[Din[d+r], 2*e+1, esize];
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.

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			

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;
if size == '11' then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = 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				

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;
if size == '11' then UNDEFINED;
constant esize = 8 << UInt(size); constant elements = 64 DIV esize;
constant d = UInt(D:Vd); constant n = UInt(N:Vn); constant m = UInt(M:Vm);
constant regs = if Q == '0' then 1 else 2;
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: 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	8
01	16
10	32
- <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.
- <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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            if !IsZero(Elem[D[n+r],e,esize] AND Elem[D[m+r],e,esize]) then
                Elem[D[d+r],e,esize] = Ones(esize);
            else
                Elem[D[d+r],e,esize] = Zeros(esize);
```

Operational information

- If CPSR.DIT is 1 and this instruction passes its condition execution check:
- The execution time of this instruction is independent of:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.
  - The response of this instruction to asynchronous exceptions does not vary based on:
    - The values of the data supplied in any of its registers.
    - The values of the NZCV flags.



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			
																												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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean signed = (U=='0');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<3:0>);
constant integer index = UInt(M);
constant integer esize = 32;
constant integer regs = 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					
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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_DotProd) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean signed = (U=='0');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm<3:0>);
constant integer index = UInt(M);
constant integer esize = 32;
constant integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

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

Operation

```
bits(64) operand1;
constant bits(64) operand2 = D[m];
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * index + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```



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					
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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant boolean signed = U=='0';
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer esize = 32;
constant integer regs = 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					
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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_DotProd) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant boolean signed = U=='0';
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer esize = 32;
constant integer regs = if Q == '1' then 2 else 1;
```

## 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.
<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

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
CheckAdvSIMDEnabled();
for r = 0 to regs-1
    operand1 = D[n+r];
    operand2 = D[m+r];
    result = D[d+r];
    integer element1, element2;
    for e = 0 to 1
        integer res = 0;
        for i = 0 to 3
            if signed then
                element1 = SInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = SInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            else
                element1 = UInt(Elem[operand1, 4 * e + i, esize DIV 4]);
                element2 = UInt(Elem[operand2, 4 * e + i, esize DIV 4]);
            res = res + element1 * element2;
        Elem[result, e, esize] = Elem[result, e, esize] + res;
    D[d+r] = result;
```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 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)

31	30	29	28	27	26	25	24	23	22	21	20	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	0	N	1	M	1	Vm			
B												U																			

Encoding for the A1 variant

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
```

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	0	D	1	0	Vn			Vd			1	1	0	0	N	1	M	1	Vm					
B												U																			

Encoding for the T1 variant

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = 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();
constant bits(128) operand1 = Q[n>>1];
constant bits(128) operand2 = Q[m>>1];
constant bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

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](#).18MM 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	1	0	1	D	0	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
																												U			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean op1_unsigned = (U == '0');
constant boolean op2_unsigned = (U == '1');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = 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	1	0	1	D	0	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			
																												U			

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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
constant boolean op1_unsigned = (U == '0');
constant boolean op2_unsigned = (U == '1');
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(Vm);
constant integer i = UInt(M);
constant integer regs = if Q == '1' then 2 else 1;
```

Assembler Symbols

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

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

operand2 = Din[m];
for r = 0 to regs-1
    operand1 = Din[n+r];
    result = Din[d+r];
    for e = 0 to 1
        bits(32) res = Elem[result, e, 32];
        for b = 0 to 3
            element1 = Int(Elem[operand1, 4 * e + b, 8], op1_unsigned);
            element2 = Int(Elem[operand2, 4 * i + b, 8], op2_unsigned);
            res = res + element1 * element2;
        Elem[result, e, 32] = res;
    D[d+r] = result;
```

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			

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;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = 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			

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 UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
constant integer regs = if Q == '1' then 2 else 1;
```

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.
- <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.

Operation

```
CheckAdvSIMDEnabled();
bits(64) operand1;
bits(64) operand2;
bits(64) result;

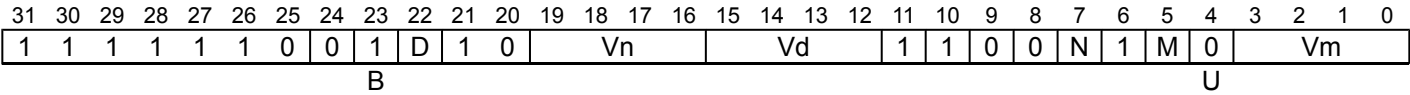
for r = 0 to regs-1
  operand1 = Din[n+r];
  operand2 = Din[m+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(32) res = Elem[result, e, 32];
    for b = 0 to 3
      constant element1 = UInt(Elem[operand1, 4 * e + b, 8]);
      constant element2 = SInt(Elem[operand2, 4 * e + b, 8]);
      res = res + element1 * element2;
    Elem[result, e, 32] = res;
  D[d+r] = result;
```



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)



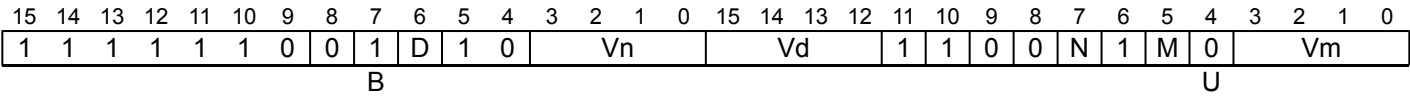
Encoding for the A1 variant

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = UInt(M:Vm);
```

T1  
(FEAT\_AA32I8MM)



Encoding for the T1 variant

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

Decode for this encoding

```
if InITBlock() then UNPREDICTABLE;
if !IsFeatureImplemented(FEAT_AA32I8MM) then UNDEFINED;
boolean op1_unsigned;
boolean op2_unsigned;
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;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
constant integer d = UInt(D:Vd);
constant integer n = UInt(N:Vn);
constant integer m = 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();
constant bits(128) operand1 = Q[n>>1];
constant bits(128) operand2 = Q[m>>1];
constant bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

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	1	0	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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant quadword_operation = (Q == '1'); constant esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant m = 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			

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant quadword_operation = (Q == '1'); constant esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	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.
- <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(); CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;
        else
            constant zipped_q = Q[m>>1]:Q[d>>1];
            for e = 0 to (128 DIV esize) - 1
                Elem[Q[d>>1],e,esize] = Elem[zipped_q,2*e,esize];
                Elem[Q[m>>1],e,esize] = Elem[zipped_q,2*e+1,esize];
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;
        else
            constant zipped_d = D[m]:D[d];
            for e = 0 to (64 DIV esize) - 1
                Elem[D[d],e,esize] = Elem[zipped_d,2*e,esize];
                Elem[D[m],e,esize] = Elem[zipped_d,2*e+1,esize];
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	size	1	0	Vd			0	0	0	0	1	0	M	0	Vm					
Q																															

Encoding for the 64-bit SIMD vector variant

VUZP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <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	size	1	0	Vd		0	0	0	0	1	0	M	0	Vm						
																														Q	

Encoding for the 64-bit SIMD vector variant

VUZP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

Assembler Symbols

- <c>

For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.

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					

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant quadword_operation = (Q == '1'); constant esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant m = 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			

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;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
constant quadword_operation = (Q == '1'); constant esize = 8 << UInt(size);
constant d = UInt(D:Vd); constant m = UInt(M:Vm);
```

Assembler Symbols

- <c> For encoding A1: see *Standard assembler syntax fields*. This encoding must be unconditional.  
For encoding T1: see *Standard assembler syntax fields*.
- <q> See *Standard assembler syntax fields*.
- <dt> For the 64-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
1x	RESERVED

For the 128-bit SIMD vector variant: is the data type for the elements of the vectors, encoded in “size”:

size	<dt>
00	8
01	16
10	32
11	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.
- <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(); CheckAdvSIMDEnabled();
    if quadword_operation then
        if d == m then
            Q[d>>1] = bits(128) UNKNOWN;
        else
            bits(256) zipped_q;
            for e = 0 to (128 DIV esize) - 1
                Elem[zipped_q,2*e,esize] = Elem[Q[d>>1],e,esize];
                Elem[zipped_q,2*e+1,esize] = Elem[Q[m>>1],e,esize];
            Q[d>>1] = zipped_q<127:0>; Q[m>>1] = zipped_q<255:128>;
    else
        if d == m then
            D[d] = bits(64) UNKNOWN;
        else
            bits(128) zipped_d;
            for e = 0 to (64 DIV esize) - 1
                Elem[zipped_d,2*e,esize] = Elem[D[d],e,esize];
                Elem[zipped_d,2*e+1,esize] = Elem[D[m],e,esize];
            D[d] = zipped_d<63:0>; D[m] = zipped_d<127:64>;
```

## Operational information

If CPSR.DIT is 1 and this instruction passes its condition execution check:

- The execution time of this instruction is independent of:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.
- The response of this instruction to asynchronous exceptions does not vary based on:
  - The values of the data supplied in any of its registers.
  - The values of the NZCV flags.

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	size	1	0	Vd				0	0	0	0	1	0	M	0	Vm				
Q																															

Encoding for the 64-bit SIMD vector variant

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <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	size	1	0	Vd				0	0	0	0	1	0	M	0	Vm				
Q																															

Encoding for the 64-bit SIMD vector variant

VZIP{<c>}{<q>}.32 <Dd>, <Dm>

is equivalent to

VTRN{<c>}{<q>}.32 <Dd>, <Dm>

Assembler Symbols

- <c>For encoding A1: see [Standard assembler syntax fields](#). This encoding must be unconditional.  
For encoding T1: 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.

## 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		<a href="#">Data-processing and miscellaneous instructions</a>			
!= 1111	010		<a href="#">Load/Store Word, Unsigned Byte (immediate, literal)</a>			
!= 1111	011	0	<a href="#">Load/Store Word, Unsigned Byte (register)</a>			
!= 1111	011	1	<a href="#">Media instructions</a>			
	10x		<a href="#">Branch, branch with link, and block data transfer</a>			
	11x		<a href="#">System register access, Advanced SIMD, floating-point, and Supervisor call</a>			
1111	0xx		<a href="#">Unconditional instructions</a>			

### 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				00		op0	op1																		op2	op3	op4				

Decode fields					Instruction details	
op0	op1	op2	op3	op4		
0		1	!= 00	1	<a href="#">Extra load/store</a>	
0	0xxxx	1	00	1	<a href="#">Multiply and Accumulate</a>	
0	1xxxx	1	00	1	<a href="#">Synchronization primitives and Load-Acquire/Store-Release</a>	
0	10xx0	0			<a href="#">Miscellaneous</a>	
0	10xx0	1		0	<a href="#">Halfword Multiply and Accumulate</a>	
0	!= 10xx0			0	<a href="#">Data-processing register (immediate shift)</a>	
0	!= 10xx0	0		1	<a href="#">Data-processing register (register shift)</a>	
1					<a href="#">Data-processing immediate</a>	

### 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				000			op0																		1	!= 00	1				

Decode fields		Instruction details	
op0			
0		<a href="#">Load/Store Dual, Half, Signed Byte (register)</a>	
1		<a href="#">Load/Store Dual, Half, Signed Byte (immediate, literal)</a>	

### 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 && cond != 1111 && op2 != 00

Decode fields				Instruction Details
P	W	o1	op2	
0	0	0	01	<a href="#">STRH (register) — post-indexed</a>
0	0	0	10	<a href="#">LDRD (register) — post-indexed</a>
0	0	0	11	<a href="#">STRD (register) — post-indexed</a>
0	0	1	01	<a href="#">LDRH (register) — post-indexed</a>
0	0	1	10	<a href="#">LDRSB (register) — post-indexed</a>
0	0	1	11	<a href="#">LDRSH (register) — post-indexed</a>
0	1	0	01	<a href="#">STRHT</a>
0	1	0	10	UNALLOCATED
0	1	0	11	UNALLOCATED
0	1	1	01	<a href="#">LDRHT</a>
0	1	1	10	<a href="#">LDRSBT</a>
0	1	1	11	<a href="#">LDRSHT</a>
1		0	01	<a href="#">STRH (register) — pre-indexed</a>
1		0	10	<a href="#">LDRD (register) — pre-indexed</a>
1		0	11	<a href="#">STRD (register) — pre-indexed</a>
1		1	01	<a href="#">LDRH (register) — pre-indexed</a>
1		1	10	<a href="#">LDRSB (register) — pre-indexed</a>
1		1	11	<a href="#">LDRSH (register) — pre-indexed</a>

### 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 && cond != 1111 && op2 != 00

Decode fields				Instruction Details
P:W	o1	Rn	op2	
	0	1111	10	<a href="#">LDRD (literal)</a>
!= 01	1	1111	01	<a href="#">LDRH (literal)</a>
!= 01	1	1111	10	<a href="#">LDRSB (literal)</a>
!= 01	1	1111	11	<a href="#">LDRSH (literal)</a>
00	0	!= 1111	10	<a href="#">LDRD (immediate) — post-indexed</a>
00	0		01	<a href="#">STRH (immediate) — post-indexed</a>
00	0		11	<a href="#">STRD (immediate) — post-indexed</a>
00	1	!= 1111	01	<a href="#">LDRH (immediate) — post-indexed</a>
00	1	!= 1111	10	<a href="#">LDRSB (immediate) — post-indexed</a>
00	1	!= 1111	11	<a href="#">LDRSH (immediate) — post-indexed</a>
01	0	!= 1111	10	UNALLOCATED
01	0		01	<a href="#">STRHT</a>
01	0		11	UNALLOCATED
01	1		01	<a href="#">LDRHT</a>
01	1		10	<a href="#">LDRSBT</a>
01	1		11	<a href="#">LDRSHT</a>
10	0	!= 1111	10	<a href="#">LDRD (immediate) — offset</a>
10	0		01	<a href="#">STRH (immediate) — offset</a>

Decode fields		Instruction Details		
P:W	o1	Rn	op2	
10	0		11	<a href="#">STRD (immediate) — offset</a>
10	1	!= 1111	01	<a href="#">LDRH (immediate) — offset</a>
10	1	!= 1111	10	<a href="#">LDRSB (immediate) — offset</a>
10	1	!= 1111	11	<a href="#">LDRSH (immediate) — offset</a>
11	0	!= 1111	10	<a href="#">LDRD (immediate) — pre-indexed</a>
11	0		01	<a href="#">STRH (immediate) — pre-indexed</a>
11	0		11	<a href="#">STRD (immediate) — pre-indexed</a>
11	1	!= 1111	01	<a href="#">LDRH (immediate) — pre-indexed</a>
11	1	!= 1111	10	<a href="#">LDRSB (immediate) — pre-indexed</a>
11	1	!= 1111	11	<a href="#">LDRSH (immediate) — pre-indexed</a>

## 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 && cond != 1111

Decode fields		Instruction Details	
opc	S		
000		<a href="#">MUL, MULS</a>	
001		<a href="#">MLA, MLAS</a>	
010	0	<a href="#">UMAAL</a>	
010	1	UNALLOCATED	
011	0	<a href="#">MLS</a>	
011	1	UNALLOCATED	
100		<a href="#">UMULL, UMULLS</a>	
101		<a href="#">UMLAL, UMLALS</a>	
110		<a href="#">SMULL, SMULLS</a>	
111		<a href="#">SMLAL, SMLALS</a>	

## 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				0001				op0													11						1001				

Decode fields		Instruction details	
op0			
0		UNALLOCATED	
1		<a href="#">Load/Store Exclusive and Load-Acquire/Store-Release</a>	

## 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 && cond != 1111

Decode fields				Instruction Details
size	L	ex	ord	
00	0	0	0	<a href="#">STL</a>
00	0	0	1	UNALLOCATED
00	0	1	0	<a href="#">STLEX</a>
00	0	1	1	<a href="#">STREX</a>
00	1	0	0	<a href="#">LDA</a>
00	1	0	1	UNALLOCATED
00	1	1	0	<a href="#">LDAEX</a>
00	1	1	1	<a href="#">LDREX</a>
01	0	0		UNALLOCATED
01	0	1	0	<a href="#">STLEXD</a>
01	0	1	1	<a href="#">STREXD</a>
01	1	0		UNALLOCATED
01	1	1	0	<a href="#">LDAEXD</a>
01	1	1	1	<a href="#">LDREXD</a>
10	0	0	0	<a href="#">STLB</a>
10	0	0	1	UNALLOCATED
10	0	1	0	<a href="#">STLEXB</a>
10	0	1	1	<a href="#">STREXB</a>
10	1	0	0	<a href="#">LDAB</a>
10	1	0	1	UNALLOCATED
10	1	1	0	<a href="#">LDAEXB</a>
10	1	1	1	<a href="#">LDREXB</a>
11	0	0	0	<a href="#">STLH</a>
11	0	0	1	UNALLOCATED
11	0	1	0	<a href="#">STLEXH</a>
11	0	1	1	<a href="#">STREXH</a>
11	1	0	0	<a href="#">LDAH</a>
11	1	0	1	UNALLOCATED
11	1	1	0	<a href="#">LDAEXH</a>
11	1	1	1	<a href="#">LDREXH</a>

## 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				00010				op0				0											0	op1							

Decode fields		Instruction details
op0	op1	
00	001	UNALLOCATED
00	010	UNALLOCATED
00	011	UNALLOCATED
00	110	UNALLOCATED

01	001	<a href="#">BX</a>
01	010	<a href="#">BXJ</a>
01	011	<a href="#">BLX (register)</a>
01	110	UNALLOCATED
10	001	UNALLOCATED
10	010	UNALLOCATED
10	011	UNALLOCATED
10	110	UNALLOCATED
11	001	<a href="#">CLZ</a>
11	010	UNALLOCATED
11	011	UNALLOCATED
11	110	<a href="#">ERET</a>
	111	<a href="#">Exception Generation</a>
	000	<a href="#">Move special register (register)</a>
	100	<a href="#">Cyclic Redundancy Check</a>
	101	<a href="#">Integer Saturating Arithmetic</a>

## 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 && cond != 1111

Decode fields opc	Instruction Details
00	<a href="#">HLT</a>
01	<a href="#">BKPT</a>
10	<a href="#">HVC</a>
11	<a href="#">SMC</a>

## 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 && cond != 1111

Decode fields opc	B	Instruction Details
x0	0	<a href="#">MRS</a>
x0	1	<a href="#">MRS (Banked register)</a>
x1	0	<a href="#">MSR (register)</a>
x1	1	<a href="#">MSR (Banked register)</a>

## 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 && cond != 1111

Decode fields		Instruction Details	Feature
sz	C		
00	0	<a href="#">CRC32</a> — <a href="#">CRC32B</a>	FEAT_CRC32
00	1	<a href="#">CRC32C</a> — <a href="#">CRC32CB</a>	FEAT_CRC32
01	0	<a href="#">CRC32</a> — <a href="#">CRC32H</a>	FEAT_CRC32
01	1	<a href="#">CRC32C</a> — <a href="#">CRC32CH</a>	FEAT_CRC32
10	0	<a href="#">CRC32</a> — <a href="#">CRC32W</a>	FEAT_CRC32
10	1	<a href="#">CRC32C</a> — <a href="#">CRC32CW</a>	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 && cond != 1111

Decode fields		Instruction Details
opc		
00		<a href="#">QADD</a>
01		<a href="#">QSUB</a>
10		<a href="#">QDADD</a>
11		<a href="#">QDSUB</a>

## 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 && cond != 1111

Decode fields			Instruction Details
opc	M	N	
00			<a href="#">SMLABB</a> , <a href="#">SMLABT</a> , <a href="#">SMLATB</a> , <a href="#">SMLATT</a>
01	0	0	<a href="#">SMLAWB</a> , <a href="#">SMLAWT</a> — <a href="#">SMLAWB</a>
01	0	1	<a href="#">SMULWB</a> , <a href="#">SMULWT</a> — <a href="#">SMULWB</a>



Decode fields			Instruction Details
opc	M	N	
01	1	0	<a href="#">SMLAWB, SMLAWT</a> — <a href="#">SMLAWT</a>
01	1	1	<a href="#">SMULWB, SMULWT</a> — <a href="#">SMULWT</a>
10			<a href="#">SMLALBB, SMLALBT, SMLALTB, SMLALTT</a>
11			<a href="#">SMULBB, SMULBT, SMULTB, SMULTT</a>

## 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				000			op0				op1																	0			

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields		Instruction details
op0	op1	
0×		<a href="#">Integer Data Processing (three register, immediate shift)</a>
10	1	<a href="#">Integer Test and Compare (two register, immediate shift)</a>
11		<a href="#">Logical Arithmetic (three register, immediate shift)</a>

## 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			stype		0	Rm							
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
000			!= 0000011	<a href="#">AND, ANDS (register)</a> — <a href="#">shift or rotate by value</a>
000			0000011	<a href="#">AND, ANDS (register)</a> — <a href="#">rotate right with extend</a>
001			!= 0000011	<a href="#">EOR, EORS (register)</a> — <a href="#">shift or rotate by value</a>
001			0000011	<a href="#">EOR, EORS (register)</a> — <a href="#">rotate right with extend</a>
010	0	!= 1101	!= 0000011	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUB, shift or rotate by value</a>
010	0	!= 1101	0000011	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUB, rotate right with extend</a>
010	0	1101	!= 0000011	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUB, shift or rotate by value</a>
010	0	1101	0000011	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUB, rotate right with extend</a>
010	1	!= 1101	!= 0000011	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUBS, shift or rotate by value</a>
010	1	!= 1101	0000011	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUBS, rotate right with extend</a>
010	1	1101	!= 0000011	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUBS, shift or rotate by value</a>
010	1	1101	0000011	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUBS, rotate right with extend</a>
011			!= 0000011	<a href="#">RSB, RSBS (register)</a> — <a href="#">shift or rotate by value</a>
011			0000011	<a href="#">RSB, RSBS (register)</a> — <a href="#">rotate right with extend</a>
100	0	!= 1101	!= 0000011	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADD, shift or rotate by value</a>
100	0	!= 1101	0000011	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADD, rotate right with extend</a>
100	0	1101	!= 0000011	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADD, shift or rotate by value</a>
100	0	1101	0000011	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADD, rotate right with extend</a>

Decode fields				Instruction Details
opc	S	Rn	imm5:type	
100	1	!= 1101	!= 0000011	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADDS, shift or rotate by value</a>
100	1	!= 1101	0000011	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADDS, rotate right with extend</a>
100	1	1101	!= 0000011	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADDS, shift or rotate by value</a>
100	1	1101	0000011	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADDS, rotate right with extend</a>
101			!= 0000011	<a href="#">ADC, ADCS (register)</a> — <a href="#">shift or rotate by value</a>
101			0000011	<a href="#">ADC, ADCS (register)</a> — <a href="#">rotate right with extend</a>
110			!= 0000011	<a href="#">SBC, SBCS (register)</a> — <a href="#">shift or rotate by value</a>
110			0000011	<a href="#">SBC, SBCS (register)</a> — <a href="#">rotate right with extend</a>
111			!= 0000011	<a href="#">RSC, RSCS (register)</a> — <a href="#">shift or rotate by value</a>
111			0000011	<a href="#">RSC, RSCS (register)</a> — <a href="#">rotate right with extend</a>

### 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 && cond != 1111

Decode fields		Instruction Details
opc	imm5:type	
00	!= 0000011	<a href="#">TST (register)</a> — <a href="#">shift or rotate by value</a>
00	0000011	<a href="#">TST (register)</a> — <a href="#">rotate right with extend</a>
01	!= 0000011	<a href="#">TEQ (register)</a> — <a href="#">shift or rotate by value</a>
01	0000011	<a href="#">TEQ (register)</a> — <a href="#">rotate right with extend</a>
10	!= 0000011	<a href="#">CMP (register)</a> — <a href="#">shift or rotate by value</a>
10	0000011	<a href="#">CMP (register)</a> — <a href="#">rotate right with extend</a>
11	!= 0000011	<a href="#">CMN (register)</a> — <a href="#">shift or rotate by value</a>
11	0000011	<a href="#">CMN (register)</a> — <a href="#">rotate right with extend</a>

### 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												imm5		stype	0							
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:type	
00	!= 0000011	<a href="#">ORR, ORRS (register)</a> — <a href="#">shift or rotate by value</a>
00	0000011	<a href="#">ORR, ORRS (register)</a> — <a href="#">rotate right with extend</a>
01	!= 0000011	<a href="#">MOV, MOVS (register)</a> — <a href="#">shift or rotate by value</a>
01	0000011	<a href="#">MOV, MOVS (register)</a> — <a href="#">rotate right with extend</a>
10	!= 0000011	<a href="#">BIC, BICS (register)</a> — <a href="#">shift or rotate by value</a>
10	0000011	<a href="#">BIC, BICS (register)</a> — <a href="#">rotate right with extend</a>
11	!= 0000011	<a href="#">MVN, MVNS (register)</a> — <a href="#">shift or rotate by value</a>

Decode fields		Instruction Details
opc	imm5:type	
11	0000011	<a href="#">MVN, MVNS (register)</a> — <a href="#">rotate right with extend</a>

## 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				000			op0			op1													0			1					

The following constraints also apply to this encoding: op0:op1 != 100

Decode fields		Instruction details
op0	op1	
0x		<a href="#">Integer Data Processing (three register, register shift)</a>
10	1	<a href="#">Integer Test and Compare (two register, register shift)</a>
11		<a href="#">Logical Arithmetic (three register, register shift)</a>

## 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 && cond != 1111

Decode fields		Instruction Details
opc		
000		<a href="#">AND, ANDS (register-shifted register)</a>
001		<a href="#">EOR, EORS (register-shifted register)</a>
010		<a href="#">SUB, SUBS (register-shifted register)</a>
011		<a href="#">RSB, RSBS (register-shifted register)</a>
100		<a href="#">ADD, ADDS (register-shifted register)</a>
101		<a href="#">ADC, ADCS (register-shifted register)</a>
110		<a href="#">SBC, SBCS (register-shifted register)</a>
111		<a href="#">RSC, RSCS (register-shifted register)</a>

## 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 && cond != 1111

Decode fields		Instruction Details
opc		
00		<a href="#">TST (register-shifted register)</a>
01		<a href="#">TEQ (register-shifted register)</a>

Decode fields	Instruction Details
opc	
10	<a href="#">CMP (register-shifted register)</a>
11	<a href="#">CMN (register-shifted register)</a>

### 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 && cond != 1111

Decode fields	Instruction Details
opc	
00	<a href="#">ORR, ORRS (register-shifted register)</a>
01	<a href="#">MOV, MOVS (register-shifted register)</a>
10	<a href="#">BIC, BICS (register-shifted register)</a>
11	<a href="#">MVN, MVNS (register-shifted register)</a>

### 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			001				op0				op1																				

Decode fields	Instruction details	
op0	op1	
0x		<a href="#">Integer Data Processing (two register and immediate)</a>
10	00	<a href="#">Move Halfword (immediate)</a>
10	10	<a href="#">Move Special Register and Hints (immediate)</a>
10	x1	<a href="#">Integer Test and Compare (one register and immediate)</a>
11		<a href="#">Logical Arithmetic (two register and immediate)</a>

### 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																
cond																															
																imm12															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details		
opc	S	Rn	
000			<a href="#">AND, ANDS (immediate)</a>
001			<a href="#">EOR, EORS (immediate)</a>
010	0	!= 11x1	<a href="#">SUB, SUBS (immediate)</a> — <a href="#">SUB</a>
010	0	1101	<a href="#">SUB, SUBS (SP minus immediate)</a> — <a href="#">SUB</a>
010	0	1111	<a href="#">ADR</a> — <a href="#">A2</a>

Decode fields			Instruction Details
opc	S	Rn	
010	1	!= 1101	<a href="#">SUB, SUBS (immediate)</a> — <a href="#">SUBS</a>
010	1	1101	<a href="#">SUB, SUBS (SP minus immediate)</a> — <a href="#">SUBS</a>
011			<a href="#">RSB, RSBS (immediate)</a>
100	0	!= 11x1	<a href="#">ADD, ADDS (immediate)</a> — <a href="#">ADD</a>
100	0	1101	<a href="#">ADD, ADDS (SP plus immediate)</a> — <a href="#">ADD</a>
100	0	1111	<a href="#">ADR</a> — <a href="#">AI</a>
100	1	!= 1101	<a href="#">ADD, ADDS (immediate)</a> — <a href="#">ADDS</a>
100	1	1101	<a href="#">ADD, ADDS (SP plus immediate)</a> — <a href="#">ADDS</a>
101			<a href="#">ADC, ADCS (immediate)</a>
110			<a href="#">SBC, SBCS (immediate)</a>
111			<a href="#">RSC, RSCS (immediate)</a>

### 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 && cond != 1111`

Decode fields	Instruction Details
H 0	<a href="#">MOV, MOVS (immediate)</a>
1	<a href="#">MOVT</a>

### 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)	imm12												
cond																															

The following constraints also apply to this encoding: `cond != 1111 && cond != 1111`

Decode fields		Instruction Details	Feature
R:imm4	imm12		
!= 00000		<a href="#">MSR (immediate)</a>	-
00000	xxxx00000000	<a href="#">NOP</a>	-
00000	xxxx00000001	<a href="#">YIELD</a>	-
00000	xxxx00000010	<a href="#">WFE</a>	-
00000	xxxx00000011	<a href="#">WFI</a>	-
00000	xxxx00000100	<a href="#">SEV</a>	-
00000	xxxx00000101	<a href="#">SEVL</a>	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP	-
00000	xxxx00010000	<a href="#">ESB</a>	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	<a href="#">TSB</a>	FEAT_TRF

Decode fields		Instruction Details	Feature
R:imm4	imm12		
00000	xxxx00010011	Reserved hint, behaves as NOP	-
00000	xxxx00010100	<a href="#">CSDB</a>	-
00000	xxxx00010101	Reserved hint, behaves as NOP	-
00000	xxxx00010110	<a href="#">CLRBHB</a>	FEAT_CLRBHB
00000	xxxx00010111	Reserved hint, behaves as NOP	-
00000	xxxx00011xxx	Reserved hint, behaves as NOP	-
00000	xxxx001xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx01xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx10xxxxxx	Reserved hint, behaves as NOP	-
00000	xxxx110xxxxx	Reserved hint, behaves as NOP	-
00000	xxxx1110xxxx	Reserved hint, behaves as NOP	-
00000	xxxx1111xxxx	<a href="#">DBG</a>	-

### 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 && cond != 1111

Decode fields opc	Instruction Details
00	<a href="#">TST (immediate)</a>
01	<a href="#">TEQ (immediate)</a>
10	<a href="#">CMP (immediate)</a>
11	<a href="#">CMN (immediate)</a>

### 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 && cond != 1111

Decode fields opc	Instruction Details
00	<a href="#">ORR, ORRS (immediate)</a>
01	<a href="#">MOV, MOVS (immediate)</a>
10	<a href="#">BIC, BICS (immediate)</a>
11	<a href="#">MVN, MVNS (immediate)</a>

### 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	o2	W	o1	Rn				Rt				imm12												
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
P:W	o2	o1	Rn	
!= 01	0	1	1111	<a href="#">LDR (literal)</a>
!= 01	1	1	1111	<a href="#">LDRB (literal)</a>
00	0	0		<a href="#">STR (immediate)</a> — <a href="#">post-indexed</a>
00	0	1	!= 1111	<a href="#">LDR (immediate)</a> — <a href="#">post-indexed</a>
00	1	0		<a href="#">STRB (immediate)</a> — <a href="#">post-indexed</a>
00	1	1	!= 1111	<a href="#">LDRB (immediate)</a> — <a href="#">post-indexed</a>
01	0	0		<a href="#">STRT</a>
01	0	1		<a href="#">LDRT</a>
01	1	0		<a href="#">STRBT</a>
01	1	1		<a href="#">LDRBT</a>
10	0	0		<a href="#">STR (immediate)</a> — <a href="#">offset</a>
10	0	1	!= 1111	<a href="#">LDR (immediate)</a> — <a href="#">offset</a>
10	1	0		<a href="#">STRB (immediate)</a> — <a href="#">offset</a>
10	1	1	!= 1111	<a href="#">LDRB (immediate)</a> — <a href="#">offset</a>
11	0	0		<a href="#">STR (immediate)</a> — <a href="#">pre-indexed</a>
11	0	1	!= 1111	<a href="#">LDR (immediate)</a> — <a href="#">pre-indexed</a>
11	1	0		<a href="#">STRB (immediate)</a> — <a href="#">pre-indexed</a>
11	1	1	!= 1111	<a href="#">LDRB (immediate)</a> — <a href="#">pre-indexed</a>

### 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	o2	W	o1	Rn				Rt				imm5				stype		0	Rm				
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields				Instruction Details
P	o2	W	o1	
0	0	0	0	<a href="#">STR (register)</a> — <a href="#">post-indexed</a>
0	0	0	1	<a href="#">LDR (register)</a> — <a href="#">post-indexed</a>
0	0	1	0	<a href="#">STRT</a>
0	0	1	1	<a href="#">LDRT</a>
0	1	0	0	<a href="#">STRB (register)</a> — <a href="#">post-indexed</a>
0	1	0	1	<a href="#">LDRB (register)</a> — <a href="#">post-indexed</a>
0	1	1	0	<a href="#">STRBT</a>
0	1	1	1	<a href="#">LDRBT</a>
1	0		0	<a href="#">STR (register)</a> — <a href="#">pre-indexed</a>
1	0		1	<a href="#">LDR (register)</a> — <a href="#">pre-indexed</a>
1	1		0	<a href="#">STRB (register)</a> — <a href="#">pre-indexed</a>
1	1		1	<a href="#">LDRB (register)</a> — <a href="#">pre-indexed</a>

### 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				011				op0														op1				1					

Decode fields		Instruction details
op0	op1	
00xxx		<a href="#">Parallel Arithmetic</a>
01000	101	<a href="#">SEL</a>
01000	001	UNALLOCATED
01000	xx0	<a href="#">PKHBT, PKHTB</a>
01001	x01	UNALLOCATED
01001	xx0	UNALLOCATED
0110x	x01	UNALLOCATED
0110x	xx0	UNALLOCATED
01x10	001	<a href="#">Saturate 16-bit</a>
01x10	101	UNALLOCATED
01x11	x01	<a href="#">Reverse Bit/Byte</a>
01x1x	xx0	<a href="#">Saturate 32-bit</a>
01xxx	111	UNALLOCATED
01xxx	011	<a href="#">Extend and Add</a>
10xxx		<a href="#">Signed multiply, Divide</a>
11000	000	<a href="#">Unsigned Sum of Absolute Differences</a>
11000	100	UNALLOCATED
11001	x00	UNALLOCATED
1101x	x00	UNALLOCATED
110xx	111	UNALLOCATED
1110x	111	UNALLOCATED
1110x	x00	<a href="#">Bitfield Insert</a>
11110	111	UNALLOCATED
11111	111	<a href="#">Permanently UNDEFINED</a>
1111x	x00	UNALLOCATED
11x0x	x10	UNALLOCATED
11x1x	x10	<a href="#">Bitfield Extract</a>
11xxx	011	UNALLOCATED
11xxx	x01	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 && cond != 1111

Decode fields			Instruction Details
op1	B	op2	
000			UNALLOCATED
001	0	00	<a href="#">SADD16</a>
001	0	01	<a href="#">SASX</a>
001	0	10	<a href="#">SSAX</a>
001	0	11	<a href="#">SSUB16</a>
001	1	00	<a href="#">SADD8</a>
001	1	01	UNALLOCATED



Decode fields			Instruction Details
op1	B	op2	
001	1	10	UNALLOCATED
001	1	11	<a href="#">SSUB8</a>
010	0	00	<a href="#">QADD16</a>
010	0	01	<a href="#">QASX</a>
010	0	10	<a href="#">QSAX</a>
010	0	11	<a href="#">QSUB16</a>
010	1	00	<a href="#">QADD8</a>
010	1	01	UNALLOCATED
010	1	10	UNALLOCATED
010	1	11	<a href="#">QSUB8</a>
011	0	00	<a href="#">SHADD16</a>
011	0	01	<a href="#">SHASX</a>
011	0	10	<a href="#">SHSAX</a>
011	0	11	<a href="#">SHSUB16</a>
011	1	00	<a href="#">SHADD8</a>
011	1	01	UNALLOCATED
011	1	10	UNALLOCATED
011	1	11	<a href="#">SHSUB8</a>
100			UNALLOCATED
101	0	00	<a href="#">UADD16</a>
101	0	01	<a href="#">UASX</a>
101	0	10	<a href="#">USAX</a>
101	0	11	<a href="#">USUB16</a>
101	1	00	<a href="#">UADD8</a>
101	1	01	UNALLOCATED
101	1	10	UNALLOCATED
101	1	11	<a href="#">USUB8</a>
110	0	00	<a href="#">UQADD16</a>
110	0	01	<a href="#">UQASX</a>
110	0	10	<a href="#">UQSAX</a>
110	0	11	<a href="#">UQSUB16</a>
110	1	00	<a href="#">UQADD8</a>
110	1	01	UNALLOCATED
110	1	10	UNALLOCATED
110	1	11	<a href="#">UQSUB8</a>
111	0	00	<a href="#">UHADD16</a>
111	0	01	<a href="#">UHASX</a>
111	0	10	<a href="#">UHSAX</a>
111	0	11	<a href="#">UHSUB16</a>
111	1	00	<a href="#">UHADD8</a>
111	1	01	UNALLOCATED
111	1	10	UNALLOCATED
111	1	11	<a href="#">UHSUB8</a>

### 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 && cond != 1111

Decode fields	Instruction Details
U	
0	<a href="#">SSAT16</a>
1	<a href="#">USAT16</a>

## 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 && cond != 1111

Decode fields		Instruction Details
o1	o2	
0	0	<a href="#">REV</a>
0	1	<a href="#">REV16</a>
1	0	<a href="#">RBIT</a>
1	1	<a href="#">REVSH</a>

## 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 && cond != 1111

Decode fields	Instruction Details
U	
0	<a href="#">SSAT</a>
1	<a href="#">USAT</a>

## 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 && cond != 1111

Decode fields			Instruction Details
U	op	Rn	
0	00	!= 1111	<a href="#">SXTAB16</a>

Decode fields			Instruction Details
U	op	Rn	
0	00	1111	<a href="#">SXTB16</a>
0	10	!= 1111	<a href="#">SXTAB</a>
0	10	1111	<a href="#">SXTB</a>
0	11	!= 1111	<a href="#">SXTAH</a>
0	11	1111	<a href="#">SXTH</a>
1	00	!= 1111	<a href="#">UXTAB16</a>
1	00	1111	<a href="#">UXTB16</a>
1	10	!= 1111	<a href="#">UXTAB</a>
1	10	1111	<a href="#">UXTB</a>
1	11	!= 1111	<a href="#">UXTAH</a>
1	11	1111	<a href="#">UXTH</a>

### 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 && cond != 1111

Decode fields			Instruction Details
op1	Ra	op2	
000	!= 1111	000	<a href="#">SMLAD, SMLADX</a> — <a href="#">SMLAD</a>
000	!= 1111	001	<a href="#">SMLAD, SMLADX</a> — <a href="#">SMLADX</a>
000	!= 1111	010	<a href="#">SMLSD, SMLSDX</a> — <a href="#">SMLSD</a>
000	!= 1111	011	<a href="#">SMLSD, SMLSDX</a> — <a href="#">SMLSDX</a>
000		1xx	UNALLOCATED
000	1111	000	<a href="#">SMUAD, SMUADX</a> — <a href="#">SMUAD</a>
000	1111	001	<a href="#">SMUAD, SMUADX</a> — <a href="#">SMUADX</a>
000	1111	010	<a href="#">SMUSD, SMUSDX</a> — <a href="#">SMUSD</a>
000	1111	011	<a href="#">SMUSD, SMUSDX</a> — <a href="#">SMUSDX</a>
001		000	<a href="#">SDIV</a>
001		!= 000	UNALLOCATED
010			UNALLOCATED
011		000	<a href="#">UDIV</a>
011		!= 000	UNALLOCATED
100		000	<a href="#">SMLALD, SMLALDX</a> — <a href="#">SMLALD</a>
100		001	<a href="#">SMLALD, SMLALDX</a> — <a href="#">SMLALDX</a>
100		010	<a href="#">SMLSLD, SMLSLDX</a> — <a href="#">SMLSLD</a>
100		011	<a href="#">SMLSLD, SMLSLDX</a> — <a href="#">SMLSLDX</a>
100		1xx	UNALLOCATED
101	!= 1111	000	<a href="#">SMMLA, SMMLAR</a> — <a href="#">SMMLA</a>
101	!= 1111	001	<a href="#">SMMLA, SMMLAR</a> — <a href="#">SMMLAR</a>
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	<a href="#">SMMLS, SMMLSR</a> — <a href="#">SMMLS</a>
101		111	<a href="#">SMMLS, SMMLSR</a> — <a href="#">SMMLSR</a>

Decode fields			Instruction Details
op1	Ra	op2	
101	1111	000	<a href="#">SMMUL, SMMULR</a> — <a href="#">SMMUL</a>
101	1111	001	<a href="#">SMMUL, SMMULR</a> — <a href="#">SMMULR</a>
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 && cond != 1111

Decode fields	Instruction Details
Ra	
!= 1111	<a href="#">USADA8</a>
1111	<a href="#">USAD8</a>

## 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 && cond != 1111

Decode fields	Instruction Details
Rn	
!= 1111	<a href="#">BFI</a>
1111	<a href="#">BFC</a>

## 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																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
cond	
0xxx	UNALLOCATED
10xx	UNALLOCATED
110x	UNALLOCATED
1110	<a href="#">UDF</a>

## 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	U	1	widthm1				Rd				lsb				1 0 1			Rn					
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
U	
0	<a href="#">SBFX</a>
1	<a href="#">UBFX</a>

## 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				10	op0																										

Decode fields		Instruction details
cond	op0	
1111	0	<a href="#">Exception Save/Restore</a>
!= 1111	0	<a href="#">Load/Store Multiple</a>
	1	<a href="#">Branch (immediate)</a>

## 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
0	0	0	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB</a> — <a href="#">Decrement After</a>
0	0	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Decrement After</a>
0	1	0	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB</a> — <a href="#">Increment After</a>
0	1	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Increment After</a>
1	0	0	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB</a> — <a href="#">Decrement Before</a>
1	0	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Decrement Before</a>
		1	1	UNALLOCATED
1	1	0	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB</a> — <a href="#">Increment Before</a>
1	1	1	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB</a> — <a href="#">Increment Before</a>

## 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 && cond != 1111

Decode fields				register_list	Instruction Details
P	U	op	L		
0	0	0	0		<a href="#">STMDA, STMED</a>
0	0	0	1		<a href="#">LMDA, LDMFA</a>
0	1	0	0		<a href="#">STM, STMIA, STMEA</a>
0	1	0	1		<a href="#">LDM, LDMIA, LDMFD</a>
		1	0		<a href="#">STM (User registers)</a>
1	0	0	0		<a href="#">STMDB, STMFD</a>
1	0	0	1		<a href="#">LDMDB, LDMEA</a>
		1	1	0xxxxxxxxxxxxxxxxxxx	<a href="#">LDM (User registers)</a>
1	1	0	0		<a href="#">STMIB, STMFA</a>
1	1	0	1		<a href="#">LDMIB, LDMED</a>
		1	1	1xxxxxxxxxxxxxxxxxxx	<a href="#">LDM (exception return)</a>

### 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			
!= 1111	0		<a href="#">B</a>
!= 1111	1		<a href="#">BL, BLX (immediate)</a> — <a href="#">A1</a>
1111			<a href="#">BL, BLX (immediate)</a> — <a href="#">A2</a>

### 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				11		op0												op1							op2						

Decode fields				Instruction details			
cond	op0	op1	op2				
	0x	0x		UNALLOCATED			
	10	0x		UNALLOCATED			
	11			<a href="#">Supervisor call</a>			
1111	!= 11	1x		<a href="#">Unconditional Advanced SIMD and floating-point instructions</a>			
!= 1111	0x	1x		<a href="#">Advanced SIMD and System register load/store and 64-bit move</a>			
!= 1111	10	1x	1	<a href="#">Advanced SIMD and System register 32-bit move</a>			
!= 1111	10	10	0	<a href="#">Floating-point data-processing</a>			
!= 1111	10	11	0	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				1111																											

Decode fields      Instruction details

cond	
1111	UNALLOCATED
!= 1111	<a href="#">SVC</a>

## 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
111111						op0			op1								1	op2	op3		op4		op5								

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			<a href="#">Advanced SIMD three registers of the same length extension</a>
100		0	!= 00	0	0	<a href="#">Floating-point conditional select</a>
101	00xxxx	0	!= 00		0	<a href="#">Floating-point minNum/maxNum</a>
101	110000	0	!= 00	1	0	<a href="#">Floating-point extraction and insertion</a>
101	111xxx	0	!= 00	1	0	<a href="#">Floating-point directed convert to integer</a>
10x		0	00			<a href="#">Advanced SIMD and floating-point multiply with accumulate</a>
10x		1	0x			<a href="#">Advanced SIMD and floating-point dot product</a>

## 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	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	<a href="#">VCADD — 64-bit SIMD vector</a>	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	<a href="#">VCADD — 128-bit SIMD vector</a>	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	<a href="#">VMMLA</a>	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	<a href="#">VDOT (vector) — 64-bit SIMD vector</a>	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	<a href="#">VDOT (vector) — 128-bit SIMD vector</a>	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	<a href="#">VFMAL (vector)</a>	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	<a href="#">VSMMLA</a>	FEAT_AA32I8MM

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
00	10	1	0	1	1	<a href="#">VUMMLA</a>	FEAT_AA32I8MM
00	10	1	1	0	0	<a href="#">VSDOT (vector)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_DotProd
00	10	1	1	0	1	<a href="#">VUDOT (vector)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_DotProd
00	10	1	1	1	0	<a href="#">VSDOT (vector)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_DotProd
00	10	1	1	1	1	<a href="#">VUDOT (vector)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_DotProd
00	11	0	0		1	<a href="#">VFMAb, VFMAbT (BFloat16, vector)</a>	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	<a href="#">VFMSL (vector)</a>	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	<a href="#">VUSMMLA</a>	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	<a href="#">VUSDOT (vector)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	<a href="#">VUSDOT (vector)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	<a href="#">VCMLA</a>	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	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														

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields		Instruction Details	Feature
size			
01		<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
10		<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — <a href="#">single-precision scalar</a>	-
11		<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — <a href="#">double-precision scalar</a>	-

## 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 && size != 00

Decode fields		Instruction Details										Feature									
size	op																				
01	0	<a href="#">VMAXNM</a> — <a href="#">half-precision scalar</a>										FEAT_FP16									
01	1	<a href="#">VMINNM</a> — <a href="#">half-precision scalar</a>										FEAT_FP16									
10	0	<a href="#">VMAXNM</a> — <a href="#">single-precision scalar</a>										-									
10	1	<a href="#">VMINNM</a> — <a href="#">single-precision scalar</a>										-									
11	0	<a href="#">VMAXNM</a> — <a href="#">double-precision scalar</a>										-									
11	1	<a href="#">VMINNM</a> — <a href="#">double-precision scalar</a>										-									

## 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 && size != 00

Decode fields		Instruction Details										Feature									
size	op																				
01		UNALLOCATED										-									
10	0	<a href="#">VMOVX</a>										FEAT_FP16									
10	1	<a href="#">VINS</a>										FEAT_FP16									
11		UNALLOCATED										-									

## 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	o1	RM	Vd			1	0	!= 00		op	1	M	0	Vm					
																	size														

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields				Instruction Details										Feature									
o1	RM	size	op																				
0		!= 00	1	UNALLOCATED										-									
0	00	01	0	<a href="#">VRINTA (floating-point)</a> — <a href="#">half-precision scalar</a>										FEAT_FP16									
0	00	10	0	<a href="#">VRINTA (floating-point)</a> — <a href="#">single-precision scalar</a>										-									
0	00	11	0	<a href="#">VRINTA (floating-point)</a> — <a href="#">double-precision scalar</a>										-									
0	01	01	0	<a href="#">VRINTN (floating-point)</a> — <a href="#">half-precision scalar</a>										FEAT_FP16									
0	01	10	0	<a href="#">VRINTN (floating-point)</a> — <a href="#">single-precision scalar</a>										-									
0	01	11	0	<a href="#">VRINTN (floating-point)</a> — <a href="#">double-precision scalar</a>										-									
0	10	01	0	<a href="#">VRINTP (floating-point)</a> — <a href="#">half-precision scalar</a>										FEAT_FP16									
0	10	10	0	<a href="#">VRINTP (floating-point)</a> — <a href="#">single-precision scalar</a>										-									
0	10	11	0	<a href="#">VRINTP (floating-point)</a> — <a href="#">double-precision scalar</a>										-									

Decode fields			Instruction Details		Feature
o1	RM	size	op		
0	11	01	0	<a href="#">VRINTM (floating-point) — half-precision scalar</a>	FEAT_FP16
0	11	10	0	<a href="#">VRINTM (floating-point) — single-precision scalar</a>	-
0	11	11	0	<a href="#">VRINTM (floating-point) — double-precision scalar</a>	-
1	00	01		<a href="#">VCVTA (floating-point) — half-precision scalar</a>	FEAT_FP16
1	00	10		<a href="#">VCVTA (floating-point) — single-precision scalar</a>	-
1	00	11		<a href="#">VCVTA (floating-point) — double-precision scalar</a>	-
1	01	01		<a href="#">VCVTN (floating-point) — half-precision scalar</a>	FEAT_FP16
1	01	10		<a href="#">VCVTN (floating-point) — single-precision scalar</a>	-
1	01	11		<a href="#">VCVTN (floating-point) — double-precision scalar</a>	-
1	10	01		<a href="#">VCVTP (floating-point) — half-precision scalar</a>	FEAT_FP16
1	10	10		<a href="#">VCVTP (floating-point) — single-precision scalar</a>	-
1	10	11		<a href="#">VCVTP (floating-point) — double-precision scalar</a>	-
1	11	01		<a href="#">VCVTM (floating-point) — half-precision scalar</a>	FEAT_FP16
1	11	10		<a href="#">VCVTM (floating-point) — single-precision scalar</a>	-
1	11	11		<a href="#">VCVTM (floating-point) — double-precision scalar</a>	-

### 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		Feature
op1	op2	Q	U			
0			0	<a href="#">VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point</a>		FEAT_FCMA
0	00		1	<a href="#">VFMAL (by scalar)</a>		FEAT_FHM
0	01		1	<a href="#">VFMSL (by scalar)</a>		FEAT_FHM
0	10		1	UNALLOCATED		-
0	11		1	<a href="#">VFMAb, VFMAbT (BFloat16, by scalar)</a>		FEAT_AA32BF16
1		0	0	<a href="#">VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point</a>		FEAT_FCMA
1			1	UNALLOCATED		-
1		1	0	<a href="#">VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point</a>		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	Q	M	U								Vm

Decode fields					Instruction Details		Feature
op1	op2	op4	Q	U			
0	00	0			UNALLOCATED		-
0	00	1	0	0	<a href="#">VDOT (by element) — 64-bit SIMD vector</a>		FEAT_AA32BF16
0	00	1		1	UNALLOCATED		-
0	00	1	1	0	<a href="#">VDOT (by element) — 128-bit SIMD vector</a>		FEAT_AA32BF16
0	01	0			UNALLOCATED		-
0	10	0			UNALLOCATED		-
0	10	1	0	0	<a href="#">VSDOT (by element) — 64-bit SIMD vector</a>		FEAT_DotProd

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	10	1	0	1	<a href="#">VUDOT (by element)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_DotProd
0	10	1	1	0	<a href="#">VSDOT (by element)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_DotProd
0	10	1	1	1	<a href="#">VUDOT (by element)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	<a href="#">VUSDOT (by element)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_AA32I8MM
1	00	1	0	1	<a href="#">VSUDOT (by element)</a> — <a href="#">64-bit SIMD vector</a>	FEAT_AA32I8MM
1	00	1	1	0	<a href="#">VUSDOT (by element)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_AA32I8MM
1	00	1	1	1	<a href="#">VSUDOT (by element)</a> — <a href="#">128-bit SIMD vector</a>	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			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, 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				110		op0															1	op1														

Decode fields		Instruction details
op0	op1	
00x0	0x	<a href="#">Advanced SIMD and floating-point 64-bit move</a>
00x0	11	<a href="#">System register 64-bit move</a>
!= 00x0	0x	<a href="#">Advanced SIMD and floating-point load/store</a>
!= 00x0	11	<a href="#">System register load/store</a>
	10	UNALLOCATED

## 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 && cond != 1111

Decode fields					Instruction Details
D	op	size	opc2	o3	
0					UNALLOCATED
1				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	<a href="#">VMOV (between two general-purpose registers and two single-precision registers)</a> — <a href="#">from general-purpose registers</a>
1	0	11	00	1	<a href="#">VMOV (between two general-purpose registers and a doubleword floating-point register)</a> — <a href="#">from general-purpose registers</a>
1			1x		UNALLOCATED
1	1	10	00	1	<a href="#">VMOV (between two general-purpose registers and two single-precision registers)</a> — <a href="#">to general-purpose registers</a>
1	1	11	00	1	<a href="#">VMOV (between two general-purpose registers and a doubleword floating-point register)</a> — <a href="#">to general-purpose registers</a>

## 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 && cond != 1111

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	<a href="#">MCRR</a>
1	1	<a href="#">MRRC</a>

## 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:U:D:W != 00x0 && cond != 1111

Decode fields				Rn	size	imm8	Instruction Details	Feature
P	U	W	L					
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">single-precision scalar</a>	-
0	1		0		11	xxxxxxxx0	<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">double-precision scalar</a>	-
0	1		0		11	xxxxxxxx1	<a href="#">FSTMDBX, FSTMIA</a> — <a href="#">Increment After</a>	-
0	1		1		10		<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">single-precision scalar</a>	-
0	1		1		11	xxxxxxxx0	<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">double-precision scalar</a>	-
0	1		1		11	xxxxxxxx1	<a href="#">FLDM*X (FLDMDBX, FLDMIA)</a> — <a href="#">Increment After</a>	-
1		0	0		01		<a href="#">VSTR</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
1		0	0		10		<a href="#">VSTR</a> — <a href="#">single-precision scalar</a>	-
1		0	0		11		<a href="#">VSTR</a> — <a href="#">double-precision scalar</a>	-
1		0	1	!= 1111	01		<a href="#">VLDR (immediate)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
1		0	1	!= 1111	10		<a href="#">VLDR (immediate)</a> — <a href="#">single-precision scalar</a>	-
1		0	1	!= 1111	11		<a href="#">VLDR (immediate)</a> — <a href="#">double-precision scalar</a>	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">single-precision scalar</a>	-
1	0	1	0		11	xxxxxxxx0	<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">double-precision scalar</a>	-
1	0	1	0		11	xxxxxxxx1	<a href="#">FSTMDBX, FSTMIA</a> — <a href="#">Decrement Before</a>	-
1	0	1	1		10		<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">single-precision scalar</a>	-
1	0	1	1		11	xxxxxxxx0	<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">double-precision scalar</a>	-
1	0	1	1		11	xxxxxxxx1	<a href="#">FLDM*X (FLDMDBX, FLDMIA)</a> — <a href="#">Decrement Before</a>	-
1		0	1	1111	01		<a href="#">VLDR (literal)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16

Decode fields						Instruction Details		Feature
P	U	W	L	Rn	size	imm8		
1		0	1	1111	10		<a href="#">VLDR (literal)</a> — <a href="#">single-precision scalar</a>	-
1		0	1	1111	11		<a href="#">VLDR (literal)</a> — <a href="#">double-precision scalar</a>	-
1	1	1					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:U:D:W != 00x0 && cond != 1111

Decode fields						Instruction Details	
P:U:W	D	L	Rn	CRd	cp15		
!= 000	0			!= 0101	0	UNALLOCATED	
!= 000	0	1	1111	0101	0	<a href="#">LDC (literal)</a>	
!= 000					1	UNALLOCATED	
!= 000	1			0101	0	UNALLOCATED	
0x1	0	0		0101	0	<a href="#">STC</a> — <a href="#">post-indexed</a>	
0x1	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">post-indexed</a>	
010	0	0		0101	0	<a href="#">STC</a> — <a href="#">unindexed</a>	
010	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">unindexed</a>	
1x0	0	0		0101	0	<a href="#">STC</a> — <a href="#">offset</a>	
1x0	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">offset</a>	
1x1	0	0		0101	0	<a href="#">STC</a> — <a href="#">pre-indexed</a>	
1x1	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">pre-indexed</a>	

## 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				1110				op0												1	op1								1				

Decode fields		Instruction details	Feature
op0	op1		
000	000	UNALLOCATED	-
000	001	<a href="#">VMOV (between general-purpose register and half-precision)</a>	FEAT_FP16
000	010	<a href="#">VMOV (between general-purpose register and single-precision)</a>	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	<a href="#">Floating-point move special register</a>	-
	011	<a href="#">Advanced SIMD 8/16/32-bit element move/duplicate</a>	-
	10x	UNALLOCATED	-
	11x	<a href="#">System register 32-bit move</a>	-

## 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 && cond != 1111

Decode fields	Instruction Details
L	
0	<a href="#">VMSR</a>
1	<a href="#">VMRS</a>

## 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 && cond != 1111

Decode fields	Instruction Details		
opc1	L	opc2	
0xx	0		<a href="#">VMOV (general-purpose register to scalar)</a>
	1		<a href="#">VMOV (scalar to general-purpose register)</a>
1xx	0	0x	<a href="#">VDUP (general-purpose register)</a>
1xx	0	1x	UNALLOCATED

## 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 && cond != 1111

Decode fields	Instruction Details
L	
0	<a href="#">MCR</a>
1	<a href="#">MRC</a>

## 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				1110				op0								10								op1								0							

Decode fields	Instruction details	
op0	op1	
1x11	1	<a href="#">Floating-point data-processing (two registers)</a>

1×11	0	<a href="#">Floating-point move immediate</a>
!= 1×11		<a href="#">Floating-point data-processing (three registers)</a>

## 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 && cond != 1111

Decode fields				Instruction Details		Feature
o1	opc2	size	o3			
		00		UNALLOCATED		-
0	000	01	0	UNALLOCATED		-
0	000	01	1	<a href="#">VABS</a> — <a href="#">half-precision scalar</a>		FEAT_FP16
0	000	10	0	<a href="#">VMOV (register)</a> — <a href="#">single-precision scalar</a>		-
0	000	10	1	<a href="#">VABS</a> — <a href="#">single-precision scalar</a>		-
0	000	11	0	<a href="#">VMOV (register)</a> — <a href="#">double-precision scalar</a>		-
0	000	11	1	<a href="#">VABS</a> — <a href="#">double-precision scalar</a>		-
0	001	01	0	<a href="#">VNEG</a> — <a href="#">half-precision scalar</a>		FEAT_FP16
0	001	01	1	<a href="#">VSQRT</a> — <a href="#">half-precision scalar</a>		FEAT_FP16
0	001	10	0	<a href="#">VNEG</a> — <a href="#">single-precision scalar</a>		-
0	001	10	1	<a href="#">VSQRT</a> — <a href="#">single-precision scalar</a>		-
0	001	11	0	<a href="#">VNEG</a> — <a href="#">double-precision scalar</a>		-
0	001	11	1	<a href="#">VSQRT</a> — <a href="#">double-precision scalar</a>		-
0	010	01		UNALLOCATED		-
0	010	10	0	<a href="#">VCVTB</a> — <a href="#">half-precision to single-precision</a>		-
0	010	10	1	<a href="#">VCVTT</a> — <a href="#">half-precision to single-precision</a>		-
0	010	11	0	<a href="#">VCVTB</a> — <a href="#">half-precision to double-precision</a>		-
0	010	11	1	<a href="#">VCVTT</a> — <a href="#">half-precision to double-precision</a>		-
0	011	01	0	<a href="#">VCVTB (BFloat16)</a>		FEAT_AA32BF16
0	011	01	1	<a href="#">VCVTT (BFloat16)</a>		FEAT_AA32BF16
0	011	10	0	<a href="#">VCVTB</a> — <a href="#">single-precision to half-precision</a>		-
0	011	10	1	<a href="#">VCVTT</a> — <a href="#">single-precision to half-precision</a>		-
0	011	11	0	<a href="#">VCVTB</a> — <a href="#">double-precision to half-precision</a>		-
0	011	11	1	<a href="#">VCVTT</a> — <a href="#">double-precision to half-precision</a>		-
0	100	01	0	<a href="#">VCMP</a>		FEAT_FP16
0	100	01	1	<a href="#">VCMPE</a>		FEAT_FP16
0	100	10	0	<a href="#">VCMP</a>		-
0	100	10	1	<a href="#">VCMPE</a>		-
0	100	11	0	<a href="#">VCMP</a>		-
0	100	11	1	<a href="#">VCMPE</a>		-
0	101	01	0	<a href="#">VCMP</a>		FEAT_FP16
0	101	01	1	<a href="#">VCMPE</a>		FEAT_FP16
0	101	10	0	<a href="#">VCMP</a>		-
0	101	10	1	<a href="#">VCMPE</a>		-
0	101	11	0	<a href="#">VCMP</a>		-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	101	11	1	<a href="#">VCMPE</a>	-
0	110	01	0	<a href="#">VRINTR</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
0	110	01	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
0	110	10	0	<a href="#">VRINTR</a> — <a href="#">single-precision scalar</a>	-
0	110	10	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">single-precision scalar</a>	-
0	110	11	0	<a href="#">VRINTR</a> — <a href="#">double-precision scalar</a>	-
0	110	11	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">double-precision scalar</a>	-
0	111	01	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">single-precision scalar</a>	-
0	111	10	1	<a href="#">VCVT</a> (between double-precision and single-precision) — <a href="#">single-precision to double-precision</a>	-
0	111	11	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">double-precision scalar</a>	-
0	111	11	1	<a href="#">VCVT</a> (between double-precision and single-precision) — <a href="#">double-precision to single-precision</a>	-
1	000	01		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
1	000	10		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">single-precision scalar</a>	-
1	000	11		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">double-precision scalar</a>	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	<a href="#">VJCVT</a>	FEAT_JSCVT
1	01x	01		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-
1	01x	11		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-
1	100	01	0	<a href="#">VCVTR</a>	FEAT_FP16
1	100	01	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	<a href="#">VCVTR</a>	-
1	100	10	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	100	11	0	<a href="#">VCVTR</a>	-
1	100	11	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	101	01	0	<a href="#">VCVTR</a>	FEAT_FP16
1	101	01	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	<a href="#">VCVTR</a>	-
1	101	10	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	101	11	0	<a href="#">VCVTR</a>	-
1	101	11	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	11x	01		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-
1	11x	11		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-

## 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 && cond != 1111



Decode fields size	Instruction Details	Feature
00	UNALLOCATED	-
01	<a href="#">VMOV (immediate)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
10	<a href="#">VMOV (immediate)</a> — <a href="#">single-precision scalar</a>	-
11	<a href="#">VMOV (immediate)</a> — <a href="#">double-precision scalar</a>	-

### 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:D:o1 != 1x11 && cond != 1111

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
!= 111	00		UNALLOCATED	-
000	01	0	<a href="#">VMLA (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
000	01	1	<a href="#">VMLS (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
000	10	0	<a href="#">VMLA (floating-point)</a> — <a href="#">single-precision scalar</a>	-
000	10	1	<a href="#">VMLS (floating-point)</a> — <a href="#">single-precision scalar</a>	-
000	11	0	<a href="#">VMLA (floating-point)</a> — <a href="#">double-precision scalar</a>	-
000	11	1	<a href="#">VMLS (floating-point)</a> — <a href="#">double-precision scalar</a>	-
001	01	0	<a href="#">VNMLS</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
001	01	1	<a href="#">VNMLA</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
001	10	0	<a href="#">VNMLS</a> — <a href="#">single-precision scalar</a>	-
001	10	1	<a href="#">VNMLA</a> — <a href="#">single-precision scalar</a>	-
001	11	0	<a href="#">VNMLS</a> — <a href="#">double-precision scalar</a>	-
001	11	1	<a href="#">VNMLA</a> — <a href="#">double-precision scalar</a>	-
010	01	0	<a href="#">VMUL (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
010	01	1	<a href="#">VNMUL</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
010	10	0	<a href="#">VMUL (floating-point)</a> — <a href="#">single-precision scalar</a>	-
010	10	1	<a href="#">VNMUL</a> — <a href="#">single-precision scalar</a>	-
010	11	0	<a href="#">VMUL (floating-point)</a> — <a href="#">double-precision scalar</a>	-
010	11	1	<a href="#">VNMUL</a> — <a href="#">double-precision scalar</a>	-
011	01	0	<a href="#">VADD (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
011	01	1	<a href="#">VSUB (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
011	10	0	<a href="#">VADD (floating-point)</a> — <a href="#">single-precision scalar</a>	-
011	10	1	<a href="#">VSUB (floating-point)</a> — <a href="#">single-precision scalar</a>	-
011	11	0	<a href="#">VADD (floating-point)</a> — <a href="#">double-precision scalar</a>	-
011	11	1	<a href="#">VSUB (floating-point)</a> — <a href="#">double-precision scalar</a>	-
100	01	0	<a href="#">VDIV</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
100	10	0	<a href="#">VDIV</a> — <a href="#">single-precision scalar</a>	-
100	11	0	<a href="#">VDIV</a> — <a href="#">double-precision scalar</a>	-
101	01	0	<a href="#">VFNMS</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
101	01	1	<a href="#">VFNMA</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
101	10	0	<a href="#">VFNMS</a> — <a href="#">single-precision scalar</a>	-
101	10	1	<a href="#">VFNMA</a> — <a href="#">single-precision scalar</a>	-

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
101	11	0	<a href="#">VFNMS</a> — <a href="#">double-precision scalar</a>	-
101	11	1	<a href="#">VFNMA</a> — <a href="#">double-precision scalar</a>	-
110	01	0	<a href="#">VFMA</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
110	01	1	<a href="#">VFMS</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
110	10	0	<a href="#">VFMA</a> — <a href="#">single-precision scalar</a>	-
110	10	1	<a href="#">VFMS</a> — <a href="#">single-precision scalar</a>	-
110	11	0	<a href="#">VFMA</a> — <a href="#">double-precision scalar</a>	-
110	11	1	<a href="#">VFMS</a> — <a href="#">double-precision scalar</a>	-

## 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
11110					op0					op1																					

Decode fields		Instruction details
op0	op1	
00x		<a href="#">Miscellaneous</a>
01x		<a href="#">Advanced SIMD data-processing</a>
1xx	1	<a href="#">Memory hints and barriers</a>
100	0	<a href="#">Advanced SIMD element or structure load/store</a>
101	0	UNALLOCATED
11x	0	UNALLOCATED

## 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
1111000							op0														op1										

Decode fields		Instruction details	Feature
op0	op1		
0xxxxx		UNALLOCATED	-
10000	xx0x	<a href="#">Change Process State</a>	-
10001	1000	UNALLOCATED	-
10001	x100	UNALLOCATED	-
10001	xx01	UNALLOCATED	-
10001	0000	<a href="#">SETPAN</a>	FEAT_PAN
1000x	0111	UNALLOCATED	-
10010	0111	CONSTRAINED UNPREDICTABLE	-
10011	0111	UNALLOCATED	-
1001x	xx0x	UNALLOCATED	-
100xx	0011	UNALLOCATED	-
100xx	0x10	UNALLOCATED	-
100xx	1x1x	UNALLOCATED	-
101xx		UNALLOCATED	-
11xxx		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)	E	A	I	F	0	mode					

Decode fields						Instruction Details	
imod	M	op	I	F	mode		
		1	0	0	0xxxx	<a href="#">SETEND</a>	
00	1	0				<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">change mode</a>	
10		0				<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt enable and change mode</a>	
		1	0	0	1xxxx	UNALLOCATED	
		1	0	1		UNALLOCATED	
		1	1			UNALLOCATED	
11		0				<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt disable and change mode</a>	

## 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	
1111001								op0																				op1				

Decode fields		Instruction details	
op0	op1		
0		<a href="#">Advanced SIMD three registers of the same length</a>	
1	0	<a href="#">Advanced SIMD two registers, or three registers of different lengths</a>	
1	1	<a href="#">Advanced SIMD shifts and immediate generation</a>	

## 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				

Decode fields					Instruction Details		Feature
U	size	opc	Q	o1			
0	0x	1100		1	<a href="#">VFMA</a>		-
0	0x	1101		0	<a href="#">VADD (floating-point)</a>		-
0	0x	1101		1	<a href="#">VMLA (floating-point)</a>		-
0	0x	1110		0	<a href="#">VCEQ (register)</a> — <a href="#">A2</a>		-
0	0x	1111		0	<a href="#">VMAX (floating-point)</a>		-
0	0x	1111		1	<a href="#">VRECPS</a>		-
		0000		0	<a href="#">VHADD</a>		-
0	00	0001		1	<a href="#">VAND (register)</a>		-
		0000		1	<a href="#">VQADD</a>		-
		0001		0	<a href="#">VRHADD</a>		-
0	00	1100		0	<a href="#">SHA1C</a>		FEAT_SHA1
		0010		0	<a href="#">VHSUB</a>		-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
0	01	0001		1	<a href="#">VBIC (register)</a>	-
		0010		1	<a href="#">VQSUB</a>	-
		0011		0	<a href="#">VCGT (register)</a> — A1	-
		0011		1	<a href="#">VCGE (register)</a> — A1	-
0	01	1100		0	<a href="#">SHA1P</a>	FEAT_SHA1
0	1x	1100		1	<a href="#">VFMS</a>	-
0	1x	1101		0	<a href="#">VSUB (floating-point)</a>	-
0	1x	1101		1	<a href="#">VMLS (floating-point)</a>	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	<a href="#">VMIN (floating-point)</a>	-
0	1x	1111		1	<a href="#">VRSQRTS</a>	-
		0100		0	<a href="#">VSHL (register)</a>	-
0		1000		0	<a href="#">VADD (integer)</a>	-
0	10	0001		1	<a href="#">VORR (register)</a>	-
0		1000		1	<a href="#">VTST</a>	-
		0100		1	<a href="#">VQSHL (register)</a>	-
0		1001		0	<a href="#">VMLA (integer)</a>	-
		0101		0	<a href="#">VRSHL</a>	-
		0101		1	<a href="#">VQRSHL</a>	-
0		1011		0	<a href="#">VQDMULH</a>	-
0	10	1100		0	<a href="#">SHA1M</a>	FEAT_SHA1
0		1011		1	<a href="#">VPADD (integer)</a>	-
		0110		0	<a href="#">VMAX (integer)</a>	-
0	11	0001		1	<a href="#">VORN (register)</a>	-
		0110		1	<a href="#">VMIN (integer)</a>	-
		0111		0	<a href="#">VABD (integer)</a>	-
		0111		1	<a href="#">VABA</a>	-
0	11	1100		0	<a href="#">SHA1SU0</a>	FEAT_SHA1
1	0x	1101		0	<a href="#">VPADD (floating-point)</a>	-
1	0x	1101		1	<a href="#">VMUL (floating-point)</a>	-
1	0x	1110		0	<a href="#">VCGE (register)</a> — A2	-
1	0x	1110		1	<a href="#">VACGE</a>	-
1	0x	1111	0	0	<a href="#">VPMAX (floating-point)</a>	-
1	0x	1111		1	<a href="#">VMAXNM</a>	-
1	00	0001		1	<a href="#">VEOR</a>	-
		1001		1	<a href="#">VMUL (integer and polynomial)</a>	-
1	00	1100		0	<a href="#">SHA256H</a>	FEAT_SHA256
		1010	0	0	<a href="#">VPMAX (integer)</a>	-
1	01	0001		1	<a href="#">VBSL</a>	-
		1010	0	1	<a href="#">VPMIN (integer)</a>	-
		1010	1		UNALLOCATED	-
1	01	1100		0	<a href="#">SHA256H2</a>	FEAT_SHA256
1	1x	1101		0	<a href="#">VABD (floating-point)</a>	-
1	1x	1110		0	<a href="#">VCGT (register)</a> — A2	-
1	1x	1110		1	<a href="#">VACGT</a>	-
1	1x	1111	0	0	<a href="#">VPMIN (floating-point)</a>	-
1	1x	1111		1	<a href="#">VMINNM</a>	-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
1		1000		0	<a href="#">VSUB (integer)</a>	-
1	10	0001		1	<a href="#">VBIT</a>	-
1		1000		1	<a href="#">VCEQ (register)</a> — <a href="#">A1</a>	-
1		1001		0	<a href="#">VMLS (integer)</a>	-
1		1011		0	<a href="#">VQRDMULH</a>	-
1	10	1100		0	<a href="#">SHA256SU1</a>	FEAT_SHA256
1		1011		1	<a href="#">VQRDMLAH</a>	FEAT_RDM
1	11	0001		1	<a href="#">VBIF</a>	-
1		1100		1	<a href="#">VQRDMLSH</a>	FEAT_RDM
1		1111	1	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
1111001							op0	1		op1							op2				op3		0								

op0	Decode fields			op3	Instruction details
	op1	op2			
0	11				<a href="#">VEXT (byte elements)</a>
1	11	0x			<a href="#">Advanced SIMD two registers misc</a>
1	11	10			<a href="#">VTBL, VTBX</a>
1	11	11			<a href="#">Advanced SIMD duplicate (scalar)</a>
	!= 11		0		<a href="#">Advanced SIMD three registers of different lengths</a>
	!= 11		1		<a href="#">Advanced SIMD two registers and a scalar</a>

## 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									

size	Decode fields			Q	Instruction Details	Feature
	opc1	opc2				
	00	0000			<a href="#">VREV64</a>	-
	00	0001			<a href="#">VREV32</a>	-
	00	0010			<a href="#">VREV16</a>	-
	00	0011			UNALLOCATED	-
	00	010x			<a href="#">VPADDL</a>	-
	00	0110	0		<a href="#">AESE</a>	FEAT_AES
	00	0110	1		<a href="#">AESD</a>	FEAT_AES
	00	0111	0		<a href="#">AESMC</a>	FEAT_AES
	00	0111	1		<a href="#">AESIMC</a>	FEAT_AES
	00	1000			<a href="#">VCLS</a>	-
00	10	0000			<a href="#">VSWP</a>	-
	00	1001			<a href="#">VCLZ</a>	-
	00	1010			<a href="#">VCNT</a>	-
	00	1011			<a href="#">VMVN (register)</a>	-

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
00	10	1100	1	UNALLOCATED	-
	00	110x		<a href="#">VPADAL</a>	-
	00	1110		<a href="#">VQABS</a>	-
	00	1111		<a href="#">VQNEG</a>	-
	01	x000		<a href="#">VCGT (immediate #0)</a>	-
	01	x001		<a href="#">VCGE (immediate #0)</a>	-
	01	x010		<a href="#">VCEQ (immediate #0)</a>	-
	01	x011		<a href="#">VCLE (immediate #0)</a>	-
	01	x100		<a href="#">VCLT (immediate #0)</a>	-
	01	x110		<a href="#">VABS</a>	-
	01	x111		<a href="#">VNEG</a>	-
	01	0101	1	<a href="#">SHA1H</a>	FEAT_SHA1
01	10	1100	1	<a href="#">VCVT (from single-precision to BFloat16, Advanced SIMD)</a>	FEAT_AA32BF16
	10	0001		<a href="#">VTRN</a>	-
	10	0010		<a href="#">VUZP</a>	-
	10	0011		<a href="#">VZIP</a>	-
	10	0100	0	<a href="#">VMOVN</a>	-
	10	0100	1	<a href="#">VQMOVN, VQMOVUN</a> — <a href="#">VQMOVUN</a>	-
	10	0101		<a href="#">VQMOVN, VQMOVUN</a> — <a href="#">VQMOVN</a>	-
	10	0110	0	<a href="#">VSHLL</a>	-
	10	0111	0	<a href="#">SHA1SU1</a>	FEAT_SHA1
	10	0111	1	<a href="#">SHA256SU0</a>	FEAT_SHA256
	10	1000		<a href="#">VRINTN (Advanced SIMD)</a>	-
	10	1001		<a href="#">VRINTX (Advanced SIMD)</a>	-
	10	1010		<a href="#">VRINTA (Advanced SIMD)</a>	-
	10	1011		<a href="#">VRINTZ (Advanced SIMD)</a>	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	<a href="#">VCVT (between half-precision and single-precision, Advanced SIMD)</a> — <a href="#">single-precision to half-precision</a>	-
	10	1101		<a href="#">VRINTM (Advanced SIMD)</a>	-
	10	1110	0	<a href="#">VCVT (between half-precision and single-precision, Advanced SIMD)</a> — <a href="#">half-precision to single-precision</a>	-
	10	1110	1	UNALLOCATED	-
	10	1111		<a href="#">VRINTP (Advanced SIMD)</a>	-
	11	000x		<a href="#">VCVTA (Advanced SIMD)</a>	-
	11	001x		<a href="#">VCVTN (Advanced SIMD)</a>	-
	11	010x		<a href="#">VCVTP (Advanced SIMD)</a>	-
	11	011x		<a href="#">VCVTM (Advanced SIMD)</a>	-
	11	10x0		<a href="#">VRECPE</a>	-
	11	10x1		<a href="#">VRSQRT</a>	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		<a href="#">VCVT (between floating-point and integer, Advanced SIMD)</a>	-

### 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		Q	M	0	Vm				

Decode fields opc	Instruction Details
000	<a href="#">VDUP (scalar)</a>
001	UNALLOCATED
01x	UNALLOCATED
1xx	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																															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields U opc	Instruction Details
	0000 <a href="#">VADDL</a>
	0001 <a href="#">VADDW</a>
	0010 <a href="#">VSUBL</a>
0	0100 <a href="#">VADDHN</a>
	0011 <a href="#">VSUBW</a>
0	0110 <a href="#">VSUBHN</a>
0	1001 <a href="#">VQDMLAL</a>
	0101 <a href="#">VABAL</a>
0	1011 <a href="#">VQDMLSL</a>
0	1101 <a href="#">VQDMULL</a>
	0111 <a href="#">VABDL (integer)</a>
	1000 <a href="#">VMLAL (integer)</a>
	1010 <a href="#">VMLSL (integer)</a>
1	0100 <a href="#">VRADDHN</a>
1	0110 <a href="#">VRSUBHN</a>
	11x0 <a href="#">VMULL (integer and polynomial)</a>
1	1001 UNALLOCATED
1	1011 UNALLOCATED
1	1101 UNALLOCATED
	1111 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 && size != 11

Decode fields Q opc	Instruction Details	Feature
	000x <a href="#">VMLA (by scalar)</a>	-

Decode fields		Instruction Details	Feature
Q	opc		
0	0011	<a href="#">VQDMLAL</a>	-
	0010	<a href="#">VMLAL (by scalar)</a>	-
0	0111	<a href="#">VQDMLSL</a>	-
	010x	<a href="#">VMLS (by scalar)</a>	-
0	1011	<a href="#">VQDMULL</a>	-
	0110	<a href="#">VMLSL (by scalar)</a>	-
	100x	<a href="#">VMUL (by scalar)</a>	-
1	0011	UNALLOCATED	-
	1010	<a href="#">VMULL (by scalar)</a>	-
1	0111	UNALLOCATED	-
	1100	<a href="#">VQDMULH</a>	-
	1101	<a href="#">VQRDMULH</a>	-
1	1011	UNALLOCATED	-
	1110	<a href="#">VQRDMLAH</a>	FEAT_RDM
	1111	<a href="#">VQRDMLSH</a>	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	
1111001								1	op0																1							

Decode fields		Instruction details
op0		
000xxxxxxxxxxxx0		<a href="#">Advanced SIMD one register and modified immediate</a>
!= 000xxxxxxxxxxxx0		<a href="#">Advanced SIMD two registers and shift amount</a>

## 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	op	
0xx0	0	<a href="#">VMOV (immediate) — A1</a>
0xx0	1	<a href="#">VMVN (immediate) — A1</a>
0xx1	0	<a href="#">VORR (immediate) — A1</a>
0xx1	1	<a href="#">VBIC (immediate) — A1</a>
10x0	0	<a href="#">VMOV (immediate) — A3</a>
10x0	1	<a href="#">VMVN (immediate) — A2</a>
10x1	0	<a href="#">VORR (immediate) — A2</a>
10x1	1	<a href="#">VBIC (immediate) — A2</a>
11xx	0	<a href="#">VMOV (immediate) — A4</a>
110x	1	<a href="#">VMVN (immediate) — A3</a>
1110	1	<a href="#">VMOV (immediate) — A5</a>
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:imm3L:Vd:opc:L != 000xxxxxxxxxxx0

Decode fields					Instruction Details
U	imm3H:L	imm3L	opc	Q	
	!= 0000		0000		<a href="#">VSHR</a>
	!= 0000		0001		<a href="#">VSRA</a>
	!= 0000	000	1010	0	<a href="#">VMOVL</a>
	!= 0000		0010		<a href="#">VRSR</a>
	!= 0000		0011		<a href="#">VRSRA</a>
	!= 0000		0111		<a href="#">VQSHL</a> , <a href="#">VQSHLU (immediate)</a> — <a href="#">VQSHL</a>
	!= 0000		1001	0	<a href="#">VQSHRN</a> , <a href="#">VQSHRUN</a> — <a href="#">VQSHRN</a>
	!= 0000		1001	1	<a href="#">VQRSHRN</a> , <a href="#">VQRSHRUN</a> — <a href="#">VQRSHRN</a>
	!= 0000		1010	0	<a href="#">VSHLL</a>
	!= 0000		11xx		<a href="#">VCVT (between floating-point and fixed-point, Advanced SIMD)</a>
0	!= 0000		0101		<a href="#">VSHL (immediate)</a>
0	!= 0000		1000	0	<a href="#">VSHRN</a>
0	!= 0000		1000	1	<a href="#">VRSHRN</a>
1	!= 0000		0100		<a href="#">VSRI</a>
1	!= 0000		0101		<a href="#">VSLI</a>
1	!= 0000		0110		<a href="#">VQSHL</a> , <a href="#">VQSHLU (immediate)</a> — <a href="#">VQSHLU</a>
1	!= 0000		1000	0	<a href="#">VQSHRN</a> , <a href="#">VQSHRUN</a> — <a href="#">VQSHRUN</a>
1	!= 0000		1000	1	<a href="#">VQRSHRN</a> , <a href="#">VQRSHRUN</a> — <a href="#">VQRSHRUN</a>

## 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
111101						op0						1													op1						

Decode fields		Instruction details
op0	op1	
00xx1		CONSTRAINED UNPREDICTABLE
01001		CONSTRAINED UNPREDICTABLE
01011		<a href="#">Barriers</a>
011x1		CONSTRAINED UNPREDICTABLE
0xxx0		<a href="#">Preload (immediate)</a>
1xxx0	0	<a href="#">Preload (register)</a>
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	Feature
opcode	option		
0000		CONSTRAINED UNPREDICTABLE	-
0001		<a href="#">CLREX</a>	-
001x		CONSTRAINED UNPREDICTABLE	-
0100	!= 0x00	<a href="#">DSB</a>	-
0100	0000	<a href="#">SSBB</a>	-
0100	0100	<a href="#">PSSBB</a>	-
0101		<a href="#">DMB</a>	-
0110		<a href="#">ISB</a>	-
0111		<a href="#">SB</a>	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		<a href="#">PLI (immediate, literal)</a>
1		1111	<a href="#">PLD (literal)</a>
1	0	!= 1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload write</a>
1	1	!= 1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload read</a>

## 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	<a href="#">PLI (register)</a> — <a href="#">shift or rotate by value</a>
0	1	0000011	<a href="#">PLI (register)</a> — <a href="#">rotate right with extend</a>
1	0	!= 0000011	<a href="#">PLD, PLDW (register)</a> — <a href="#">preload write, optional shift or rotate</a>
1	0	0000011	<a href="#">PLD, PLDW (register)</a> — <a href="#">preload write, rotate right with extend</a>
1	1	!= 0000011	<a href="#">PLD, PLDW (register)</a> — <a href="#">preload read, optional shift or rotate</a>
1	1	0000011	<a href="#">PLD, PLDW (register)</a> — <a href="#">preload read, rotate right with extend</a>

## 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
11110100								op0			0									op1											

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD load/store multiple structures</a>
1	11	<a href="#">Advanced SIMD load single structure to all lanes</a>
1	!= 11	<a href="#">Advanced SIMD load/store single structure to one lane</a>

## 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	
0	000x != 11x1	<a href="#">VST4 (multiple 4-element structures)</a>
0	000x 1101	<a href="#">VST4 (multiple 4-element structures)</a>
0	000x 1111	<a href="#">VST4 (multiple 4-element structures)</a>
0	0010 != 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0010 1101	<a href="#">VST1 (multiple single elements)</a>
0	0010 1111	<a href="#">VST1 (multiple single elements)</a>
0	0011 != 11x1	<a href="#">VST2 (multiple 2-element structures)</a>
0	0011 1101	<a href="#">VST2 (multiple 2-element structures)</a>
0	0011 1111	<a href="#">VST2 (multiple 2-element structures)</a>
0	010x != 11x1	<a href="#">VST3 (multiple 3-element structures)</a>
0	010x 1101	<a href="#">VST3 (multiple 3-element structures)</a>
0	010x 1111	<a href="#">VST3 (multiple 3-element structures)</a>
0	0110 != 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0110 1101	<a href="#">VST1 (multiple single elements)</a>
0	0110 1111	<a href="#">VST1 (multiple single elements)</a>
0	0111 != 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0111 1101	<a href="#">VST1 (multiple single elements)</a>
0	0111 1111	<a href="#">VST1 (multiple single elements)</a>
0	100x != 11x1	<a href="#">VST2 (multiple 2-element structures)</a>
0	100x 1101	<a href="#">VST2 (multiple 2-element structures)</a>
0	100x 1111	<a href="#">VST2 (multiple 2-element structures)</a>
0	1010 != 11x1	<a href="#">VST1 (multiple single elements)</a>
0	1010 1101	<a href="#">VST1 (multiple single elements)</a>
0	1010 1111	<a href="#">VST1 (multiple single elements)</a>
1	000x != 11x1	<a href="#">VLD4 (multiple 4-element structures)</a>
1	000x 1101	<a href="#">VLD4 (multiple 4-element structures)</a>
1	000x 1111	<a href="#">VLD4 (multiple 4-element structures)</a>
1	0010 != 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0010 1101	<a href="#">VLD1 (multiple single elements)</a>
1	0010 1111	<a href="#">VLD1 (multiple single elements)</a>
1	0011 != 11x1	<a href="#">VLD2 (multiple 2-element structures)</a>
1	0011 1101	<a href="#">VLD2 (multiple 2-element structures)</a>
1	0011 1111	<a href="#">VLD2 (multiple 2-element structures)</a>
1	010x != 11x1	<a href="#">VLD3 (multiple 3-element structures)</a>

Decode fields			Instruction Details
L	itype	Rm	
1	010x	1101	<a href="#">VLD3 (multiple 3-element structures)</a>
1	010x	1111	<a href="#">VLD3 (multiple 3-element structures)</a>
	1011		UNALLOCATED
1	0110	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0110	1101	<a href="#">VLD1 (multiple single elements)</a>
1	0110	1111	<a href="#">VLD1 (multiple single elements)</a>
1	0111	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0111	1101	<a href="#">VLD1 (multiple single elements)</a>
1	0111	1111	<a href="#">VLD1 (multiple single elements)</a>
	11xx		UNALLOCATED
1	100x	!= 11x1	<a href="#">VLD2 (multiple 2-element structures)</a>
1	100x	1101	<a href="#">VLD2 (multiple 2-element structures)</a>
1	100x	1111	<a href="#">VLD2 (multiple 2-element structures)</a>
1	1010	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	1010	1101	<a href="#">VLD1 (multiple single elements)</a>
1	1010	1111	<a href="#">VLD1 (multiple single elements)</a>

### 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		!= 11x1	<a href="#">VLD1 (single element to all lanes)</a>
1	00		1101	<a href="#">VLD1 (single element to all lanes)</a>
1	00		1111	<a href="#">VLD1 (single element to all lanes)</a>
1	01		!= 11x1	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	01		1101	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	01		1111	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	10	0	!= 11x1	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	0	1101	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	0	1111	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	1		UNALLOCATED
1	11		!= 11x1	<a href="#">VLD4 (single 4-element structure to all lanes)</a>
1	11		1101	<a href="#">VLD4 (single 4-element structure to all lanes)</a>
1	11		1111	<a href="#">VLD4 (single 4-element structure to all lanes)</a>

### 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																															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	00	00	1101	<a href="#">VST1 (single element from one lane)</a>
0	00	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	00	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	00	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	00	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	01	00	1101	<a href="#">VST1 (single element from one lane)</a>
0	01	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	01	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	10	00	1101	<a href="#">VST1 (single element from one lane)</a>
0	10	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	10	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
1	00	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	00	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	00	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	00	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	00	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>

Decode fields				Instruction Details
L	size	N	Rm	
1	00	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	01	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	01	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	01	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	10	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	10	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	10	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

## 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		<a href="#">16-bit</a>
111	00	<a href="#">B — T2</a>
111	!= 00	<a href="#">32-bit</a>

### 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
op0															

The following constraints also apply to this encoding: op0<5:3> != 111

Decode fields	Instruction details
op0	
00xxxx	<a href="#">Shift (immediate), add, subtract, move, and compare</a>
010000	<a href="#">Data-processing (two low registers)</a>
010001	<a href="#">Special data instructions and branch and exchange</a>
01001x	<a href="#">LDR (literal) — T1</a>
0101xx	<a href="#">Load/store (register offset)</a>
011xxx	<a href="#">Load/store word/byte (immediate offset)</a>
1000xx	<a href="#">Load/store halfword (immediate offset)</a>
1001xx	<a href="#">Load/store (SP-relative)</a>
1010xx	<a href="#">Add PC/SP (immediate)</a>
1011xx	<a href="#">Miscellaneous 16-bit instructions</a>
1100xx	<a href="#">Load/store multiple</a>
1101xx	<a href="#">Conditional branch, and Supervisor Call</a>

### 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
00	op0	op1	op2												

Decode fields			Instruction details
op0	op1	op2	
0	11	0	<a href="#">Add, subtract (three low registers)</a>
0	11	1	<a href="#">Add, subtract (two low registers and immediate)</a>
0	!= 11		<a href="#">MOV, MOVS (register) — T2</a>
1			<a href="#">Add, subtract, compare, move (one low register and immediate)</a>

### 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	Instruction Details
S	
0	<a href="#">ADD, ADDS (register)</a>
1	<a href="#">SUB, SUBS (register)</a>

### 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	Instruction Details
S	
0	<a href="#">ADD, ADDS (immediate)</a>
1	<a href="#">SUB, SUBS (immediate)</a>

### 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	Instruction Details
op	
00	<a href="#">MOV, MOVS (immediate)</a>
01	<a href="#">CMP (immediate)</a>
10	<a href="#">ADD, ADDS (immediate)</a>
11	<a href="#">SUB, SUBS (immediate)</a>

### 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	Instruction Details
op	
0000	<a href="#">AND, ANDS (register)</a>
0001	<a href="#">EOR, EORS (register)</a>
0010	<a href="#">MOV, MOVS (register-shifted register)</a> — <a href="#">logical shift left</a>
0011	<a href="#">MOV, MOVS (register-shifted register)</a> — <a href="#">logical shift right</a>
0100	<a href="#">MOV, MOVS (register-shifted register)</a> — <a href="#">arithmetic shift right</a>
0101	<a href="#">ADC, ADCS (register)</a>
0110	<a href="#">SBC, SBCS (register)</a>
0111	<a href="#">MOV, MOVS (register-shifted register)</a> — <a href="#">rotate right</a>
1000	<a href="#">TST (register)</a>
1001	<a href="#">RSB, RSBS (immediate)</a>
1010	<a href="#">CMP (register)</a>
1011	<a href="#">CMN (register)</a>



Decode fields	Instruction Details
<b>op</b>	
1100	<a href="#">ORR, ORRS (register)</a>
1101	<a href="#">MUL, MULS</a>
1110	<a href="#">BIC, BICS (register)</a>
1111	<a href="#">MVN, MVNS (register)</a>

## 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	Instruction details
<b>op0</b>	
11	<a href="#">Branch and exchange</a>
!= 11	<a href="#">Add, subtract, compare, move (two high registers)</a>

### 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	Instruction Details
<b>L</b>	
0	<a href="#">BX</a>
1	<a href="#">BLX (register)</a>

### 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						
op															

The following constraints also apply to this encoding: op != 11 && op != 11

Decode fields			Instruction Details
op	D:Rd	Rs	
00	!= 1101	!= 1101	<a href="#">ADD, ADDS (register)</a>
00		1101	<a href="#">ADD, ADDS (SP plus register) — T1</a>
00	1101	!= 1101	<a href="#">ADD, ADDS (SP plus register) — T2</a>
01			<a href="#">CMP (register)</a>
10			<a href="#">MOV, MOVS (register)</a>

### 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	<a href="#">STR (register)</a>
0	0	1	<a href="#">STRH (register)</a>
0	1	0	<a href="#">STRB (register)</a>
0	1	1	<a href="#">LDRSB (register)</a>
1	0	0	<a href="#">LDR (register)</a>
1	0	1	<a href="#">LDRH (register)</a>
1	1	0	<a href="#">LDRB (register)</a>
1	1	1	<a href="#">LDRSH (register)</a>

### 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	<a href="#">STR (immediate)</a>
0	1	<a href="#">LDR (immediate)</a>
1	0	<a href="#">STRB (immediate)</a>
1	1	<a href="#">LDRB (immediate)</a>

### 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	<a href="#">STRH (immediate)</a>
1	<a href="#">LDRH (immediate)</a>

### 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	Instruction Details
L	
0	<a href="#">STR (immediate)</a>
1	<a href="#">LDR (immediate)</a>

### 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	Instruction Details
SP	
0	<a href="#">ADR</a>
1	<a href="#">ADD, ADDS (SP plus immediate)</a>

## 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
1011	op0	op1	op2	op3											

op0	op1	op2	op3	Instruction details	Feature
0000				<a href="#">Adjust SP (immediate)</a>	-
0010				<a href="#">Extend</a>	-
0110	00	0		<a href="#">SETPAN</a>	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			<a href="#">Change Processor State</a>	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			<a href="#">HLT</a>	-
1010	!= 10			<a href="#">Reverse bytes</a>	-
1110				<a href="#">BKPT</a>	-
1111			0000	<a href="#">Hints</a>	-
1111			!= 0000	<a href="#">IT</a>	-
x0x1				<a href="#">CBNZ, CBZ</a>	-
x10x				<a href="#">Push and Pop</a>	-

### 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	Instruction Details
S	
0	<a href="#">ADD, ADDS (SP plus immediate)</a>
1	<a href="#">SUB, SUBS (SP minus immediate)</a>

### 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	<a href="#">SXTH</a>
0	1	<a href="#">SXTB</a>
1	0	<a href="#">UXTH</a>

Decode fields		Instruction Details
U	B	
1	1	<a href="#">UXTB</a>

## 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		<a href="#">SETEND</a>
1	0xxxx	<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt enable</a>
1	1xxxx	<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt disable</a>

## 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 && op != 10

Decode fields		Instruction Details
op		
00		<a href="#">REV</a>
01		<a href="#">REV16</a>
11		<a href="#">REVSH</a>

## 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		<a href="#">NOP</a>
0001		<a href="#">YIELD</a>
0010		<a href="#">WFE</a>
0011		<a href="#">WFI</a>
0100		<a href="#">SEV</a>
0101		<a href="#">SEVL</a>
011x		Reserved hint, behaves as NOP
1xxx		Reserved hint, behaves as NOP

## 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	<a href="#">PUSH</a>
1	<a href="#">POP</a>

## 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	<a href="#">STM, STMIA, STMEA</a>
1	<a href="#">LDM, LDMIA, LDMFD</a>

## 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
1101				op0											

Decode fields	Instruction details
op0	
111x	<a href="#">Exception generation</a>
!= 111x	<a href="#">B — T1</a>

## 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	<a href="#">UDF</a>
1	<a href="#">SVC</a>

## 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
111			op0				op1								op3																

The following constraints also apply to this encoding: op0<3:2> != 00

Decode fields			Instruction details
op0	op1	op3	
x11x			<a href="#">System register access, Advanced SIMD, and floating-point</a>

0100	xx0xx		<a href="#">Load/store multiple</a>
0100	xx1xx		<a href="#">Load/store dual, load/store exclusive, load-acquire/store-release, and table branch</a>
0101			<a href="#">Data-processing (shifted register)</a>
10xx		1	<a href="#">Branches and miscellaneous control</a>
10x0		0	<a href="#">Data-processing (modified immediate)</a>
10x1	xxxx0	0	<a href="#">Data-processing (plain binary immediate)</a>
10x1	xxxx1	0	UNALLOCATED
1100	1xxx0		<a href="#">Advanced SIMD element or structure load/store</a>
1100	!= 1xxx0		<a href="#">Load/store single</a>
1101	0xxxx		<a href="#">Data-processing (register)</a>
1101	10xxx		<a href="#">Multiply, multiply accumulate, and absolute difference</a>
1101	11xxx		<a href="#">Long multiply and divide</a>

## 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
111	op0	11	op1													op2										op3					

Decode fields				Instruction details
op0	op1	op2	op3	
	0x	0x		UNALLOCATED
	10	0x		UNALLOCATED
	11			<a href="#">Advanced SIMD data-processing</a>
0	0x	1x		<a href="#">Advanced SIMD and System register load/store and 64-bit move</a>
0	10	1x	1	<a href="#">Advanced SIMD and System register 32-bit move</a>
0	10	10	0	<a href="#">Floating-point data-processing</a>
0	10	11	0	UNALLOCATED
1	!= 11	1x		<a href="#">Additional Advanced SIMD and floating-point instructions</a>

## 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
111			1111	op0																							op1				

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD three registers of the same length</a>
1	0	<a href="#">Advanced SIMD two registers, or three registers of different lengths</a>
1	1	<a href="#">Advanced SIMD shifts and immediate generation</a>

## 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		Q	o1	Instruction Details	Feature
	size	opc				
0	0x	1100		1	<a href="#">VFMA</a>	-
0	0x	1101		0	<a href="#">VADD (floating-point)</a>	-
0	0x	1101		1	<a href="#">VMLA (floating-point)</a>	-
0	0x	1110		0	<a href="#">VCEQ (register) — T2</a>	-
0	0x	1111		0	<a href="#">VMAX (floating-point)</a>	-
0	0x	1111		1	<a href="#">VRECPS</a>	-
		0000		0	<a href="#">VHADD</a>	-
0	00	0001		1	<a href="#">VAND (register)</a>	-
		0000		1	<a href="#">VQADD</a>	-
		0001		0	<a href="#">VRHADD</a>	-
0	00	1100		0	<a href="#">SHA1C</a>	FEAT_SHA1
		0010		0	<a href="#">VHSUB</a>	-
0	01	0001		1	<a href="#">VBIC (register)</a>	-
		0010		1	<a href="#">VQSUB</a>	-
		0011		0	<a href="#">VCGT (register) — T1</a>	-
		0011		1	<a href="#">VCGE (register) — T1</a>	-
0	01	1100		0	<a href="#">SHA1P</a>	FEAT_SHA1
0	1x	1100		1	<a href="#">VFMS</a>	-
0	1x	1101		0	<a href="#">VSUB (floating-point)</a>	-
0	1x	1101		1	<a href="#">VMLS (floating-point)</a>	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	<a href="#">VMIN (floating-point)</a>	-
0	1x	1111		1	<a href="#">VRSQRTS</a>	-
		0100		0	<a href="#">VSHL (register)</a>	-
0		1000		0	<a href="#">VADD (integer)</a>	-
0	10	0001		1	<a href="#">VORR (register)</a>	-
0		1000		1	<a href="#">VTST</a>	-
		0100		1	<a href="#">VQSHL (register)</a>	-
0		1001		0	<a href="#">VMLA (integer)</a>	-
		0101		0	<a href="#">VRSHL</a>	-
		0101		1	<a href="#">VQRSHL</a>	-
0		1011		0	<a href="#">VQDMULH</a>	-
0	10	1100		0	<a href="#">SHA1M</a>	FEAT_SHA1
0		1011		1	<a href="#">VPADD (integer)</a>	-
		0110		0	<a href="#">VMAX (integer)</a>	-
0	11	0001		1	<a href="#">VORN (register)</a>	-
		0110		1	<a href="#">VMIN (integer)</a>	-
		0111		0	<a href="#">VABD (integer)</a>	-
		0111		1	<a href="#">VABA</a>	-
0	11	1100		0	<a href="#">SHA1SU0</a>	FEAT_SHA1
1	0x	1101		0	<a href="#">VPADD (floating-point)</a>	-
1	0x	1101		1	<a href="#">VMUL (floating-point)</a>	-
1	0x	1110		0	<a href="#">VCGE (register) — T2</a>	-
1	0x	1110		1	<a href="#">VACGE</a>	-
1	0x	1111	0	0	<a href="#">VPMAX (floating-point)</a>	-
1	0x	1111		1	<a href="#">VMAXNM</a>	-
1	00	0001		1	<a href="#">VEOR</a>	-

Decode fields					Instruction Details	Feature
U	size	opc	Q	o1		
		1001		1	<a href="#">VMUL (integer and polynomial)</a>	-
1	00	1100		0	<a href="#">SHA256H</a>	FEAT_SHA256
		1010	0	0	<a href="#">VPMAX (integer)</a>	-
1	01	0001		1	<a href="#">VBSL</a>	-
		1010	0	1	<a href="#">VPMIN (integer)</a>	-
		1010	1		UNALLOCATED	-
1	01	1100		0	<a href="#">SHA256H2</a>	FEAT_SHA256
1	1x	1101		0	<a href="#">VABD (floating-point)</a>	-
1	1x	1110		0	<a href="#">VCGT (register) — T2</a>	-
1	1x	1110		1	<a href="#">VACGT</a>	-
1	1x	1111	0	0	<a href="#">VPMIN (floating-point)</a>	-
1	1x	1111		1	<a href="#">VMINNM</a>	-
1		1000		0	<a href="#">VSUB (integer)</a>	-
1	10	0001		1	<a href="#">VBIT</a>	-
1		1000		1	<a href="#">VCEQ (register) — T1</a>	-
1		1001		0	<a href="#">VMLS (integer)</a>	-
1		1011		0	<a href="#">VQRDMULH</a>	-
1	10	1100		0	<a href="#">SHA256SU1</a>	FEAT_SHA256
1		1011		1	<a href="#">VQRDMLAH</a>	FEAT_RDM
1	11	0001		1	<a href="#">VBIF</a>	-
1		1100		1	<a href="#">VQRDMLSH</a>	FEAT_RDM
1		1111	1	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
111			op0		11111						op1							op2							op3			0			

Decode fields				Instruction details	
op0	op1	op2	op3		
0	11			<a href="#">VEXT (byte elements)</a>	
1	11	0x		<a href="#">Advanced SIMD two registers misc</a>	
1	11	10		<a href="#">VTBL, VTBX</a>	
1	11	11		<a href="#">Advanced SIMD duplicate (scalar)</a>	
	!= 11		0	<a href="#">Advanced SIMD three registers of different lengths</a>	
	!= 11		1	<a href="#">Advanced SIMD two registers and a scalar</a>	

## 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				

size	Decode fields			Q	Instruction Details	Feature
	opc1	opc2				
	00	0000			<a href="#">VREV64</a>	-
	00	0001			<a href="#">VREV32</a>	-



size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	00	0010		<a href="#">VREV16</a>	-
	00	0011		UNALLOCATED	-
	00	010x		<a href="#">VPADDL</a>	-
	00	0110	0	<a href="#">AESE</a>	FEAT_AES
	00	0110	1	<a href="#">AESD</a>	FEAT_AES
	00	0111	0	<a href="#">AESMC</a>	FEAT_AES
	00	0111	1	<a href="#">AESIMC</a>	FEAT_AES
	00	1000		<a href="#">VCLS</a>	-
00	10	0000		<a href="#">VSWP</a>	-
	00	1001		<a href="#">VCLZ</a>	-
	00	1010		<a href="#">VCNT</a>	-
	00	1011		<a href="#">VMVN (register)</a>	-
00	10	1100	1	UNALLOCATED	-
	00	110x		<a href="#">VPADAL</a>	-
	00	1110		<a href="#">VQABS</a>	-
	00	1111		<a href="#">VQNEG</a>	-
	01	x000		<a href="#">VCGT (immediate #0)</a>	-
	01	x001		<a href="#">VCGE (immediate #0)</a>	-
	01	x010		<a href="#">VCEQ (immediate #0)</a>	-
	01	x011		<a href="#">VCLE (immediate #0)</a>	-
	01	x100		<a href="#">VCLT (immediate #0)</a>	-
	01	x110		<a href="#">VABS</a>	-
	01	x111		<a href="#">VNEG</a>	-
	01	0101	1	<a href="#">SHA1H</a>	FEAT_SHA1
01	10	1100	1	<a href="#">VCVT (from single-precision to BFloat16, Advanced SIMD)</a>	FEAT_AA32BF16
	10	0001		<a href="#">VTRN</a>	-
	10	0010		<a href="#">VUZP</a>	-
	10	0011		<a href="#">VZIP</a>	-
	10	0100	0	<a href="#">VMOVN</a>	-
	10	0100	1	<a href="#">VQMOVN, VQMOVUN</a> — <a href="#">VQMOVUN</a>	-
	10	0101		<a href="#">VQMOVN, VQMOVUN</a> — <a href="#">VQMOVN</a>	-
	10	0110	0	<a href="#">VSHLL</a>	-
	10	0111	0	<a href="#">SHA1SU1</a>	FEAT_SHA1
	10	0111	1	<a href="#">SHA256SU0</a>	FEAT_SHA256
	10	1000		<a href="#">VRINTN (Advanced SIMD)</a>	-
	10	1001		<a href="#">VRINTX (Advanced SIMD)</a>	-
	10	1010		<a href="#">VRINTA (Advanced SIMD)</a>	-
	10	1011		<a href="#">VRINTZ (Advanced SIMD)</a>	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	<a href="#">VCVT (between half-precision and single-precision, Advanced SIMD)</a> — <a href="#">single-precision to half-precision</a>	-
	10	1101		<a href="#">VRINTM (Advanced SIMD)</a>	-
	10	1110	0	<a href="#">VCVT (between half-precision and single-precision, Advanced SIMD)</a> — <a href="#">half-precision to single-precision</a>	-
	10	1110	1	UNALLOCATED	-
	10	1111		<a href="#">VRINTP (Advanced SIMD)</a>	-
	11	000x		<a href="#">VCVTA (Advanced SIMD)</a>	-
	11	001x		<a href="#">VCVTN (Advanced SIMD)</a>	-

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	11	010x		<a href="#">VCVTP (Advanced SIMD)</a>	-
	11	011x		<a href="#">VCVTM (Advanced SIMD)</a>	-
	11	10x0		<a href="#">VRECPE</a>	-
	11	10x1		<a href="#">VRSQRTE</a>	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		<a href="#">VCVT (between floating-point and integer, Advanced SIMD)</a>	-

### 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	
000	<a href="#">VDUP (scalar)</a>
001	UNALLOCATED
01x	UNALLOCATED
1xx	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																															

The following constraints also apply to this encoding: size != 11 && size != 11

Decode fields	Instruction Details
U opc	
	0000 <a href="#">VADDL</a>
	0001 <a href="#">VADDW</a>
	0010 <a href="#">VSUBL</a>
0	0100 <a href="#">VADDHN</a>
	0011 <a href="#">VSUBW</a>
0	0110 <a href="#">VSUBHN</a>
0	1001 <a href="#">VQDMLAL</a>
	0101 <a href="#">VABAL</a>
0	1011 <a href="#">VQDMLSL</a>
0	1101 <a href="#">VQDMULL</a>
	0111 <a href="#">VABDL (integer)</a>
	1000 <a href="#">VMLAL (integer)</a>
	1010 <a href="#">VMLSL (integer)</a>
1	0100 <a href="#">VRADDHN</a>
1	0110 <a href="#">VRSUBHN</a>
	11x0 <a href="#">VMULL (integer and polynomial)</a>
1	1001 UNALLOCATED
1	1011 UNALLOCATED

Decode fields	Instruction Details
U opc	
1	1101 UNALLOCATED
	1111 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 && size != 11

Decode fields	Instruction Details	Feature
Q opc		
	000x	<a href="#">VMLA (by scalar)</a> -
0	0011	<a href="#">VQDMLAL</a> -
	0010	<a href="#">VMLAL (by scalar)</a> -
0	0111	<a href="#">VQDMLSL</a> -
	010x	<a href="#">VMLS (by scalar)</a> -
0	1011	<a href="#">VQDMULL</a> -
	0110	<a href="#">VMLSL (by scalar)</a> -
	100x	<a href="#">VMUL (by scalar)</a> -
1	0011	UNALLOCATED -
	1010	<a href="#">VMULL (by scalar)</a> -
1	0111	UNALLOCATED -
	1100	<a href="#">VQDMULH</a> -
	1101	<a href="#">VQDMLH</a> -
1	1011	UNALLOCATED -
	1110	<a href="#">VQRDMLAH</a> FEAT_RDM
	1111	<a href="#">VQRDMLSH</a> 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
111				11111				op0											1												

Decode fields	Instruction details
op0	
000xxxxxxxxxxx0	<a href="#">Advanced SIMD one register and modified immediate</a>
!= 000xxxxxxxxxxx0	<a href="#">Advanced SIMD two registers and shift amount</a>

## 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 cmode	op	Instruction Details
0xx0	0	<a href="#">VMOV (immediate) — T1</a>
0xx0	1	<a href="#">VMVN (immediate) — T1</a>
0xx1	0	<a href="#">VORR (immediate) — T1</a>
0xx1	1	<a href="#">VBIC (immediate) — T1</a>
10x0	0	<a href="#">VMOV (immediate) — T3</a>
10x0	1	<a href="#">VMVN (immediate) — T2</a>
10x1	0	<a href="#">VORR (immediate) — T2</a>
10x1	1	<a href="#">VBIC (immediate) — T2</a>
11xx	0	<a href="#">VMOV (immediate) — T4</a>
110x	1	<a href="#">VMVN (immediate) — T3</a>
1110	1	<a href="#">VMOV (immediate) — T5</a>
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:imm3L:Vd:opc:L != 000xxxxxxxxxx0

U	Decode fields imm3H:L	imm3L	opc	Q	Instruction Details
	!= 0000		0000		<a href="#">VSHR</a>
	!= 0000		0001		<a href="#">VSRA</a>
	!= 0000	000	1010	0	<a href="#">VMOVL</a>
	!= 0000		0010		<a href="#">VRSR</a>
	!= 0000		0011		<a href="#">VRSRA</a>
	!= 0000		0111		<a href="#">VQSHL, VQSHLU (immediate) — VQSHL</a>
	!= 0000		1001	0	<a href="#">VQSHRN, VQSHRUN — VQSHRN</a>
	!= 0000		1001	1	<a href="#">VQRSHRN, VQRSHRUN — VQRSHRN</a>
	!= 0000		1010	0	<a href="#">VSHLL</a>
	!= 0000		11xx		<a href="#">VCVT (between floating-point and fixed-point, Advanced SIMD)</a>
0	!= 0000		0101		<a href="#">VSHL (immediate)</a>
0	!= 0000		1000	0	<a href="#">VSHRN</a>
0	!= 0000		1000	1	<a href="#">VRSR</a>
1	!= 0000		0100		<a href="#">VSRI</a>
1	!= 0000		0101		<a href="#">VSLI</a>
1	!= 0000		0110		<a href="#">VQSHL, VQSHLU (immediate) — VQSHLU</a>
1	!= 0000		1000	0	<a href="#">VQSHRN, VQSHRUN — VQSHRN</a>
1	!= 0000		1000	1	<a href="#">VQRSHRN, VQRSHRUN — VQRSHRN</a>

### 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			
1110110								op0																1	op1									

Decode fields		Instruction details
op0	op1	
00x0	0x	<a href="#">Advanced SIMD and floating-point 64-bit move</a>
00x0	11	<a href="#">System register 64-bit move</a>
!= 00x0	0x	<a href="#">Advanced SIMD and floating-point load/store</a>
!= 00x0	11	<a href="#">System register Load/Store</a>
	10	UNALLOCATED

### 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				0	UNALLOCATED
1		0x	00	1	UNALLOCATED
1			01		UNALLOCATED
1	0	10	00	1	<a href="#">VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers</a>
1	0	11	00	1	<a href="#">VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers</a>
1			1x		UNALLOCATED
1	1	10	00	1	<a href="#">VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers</a>
1	1	11	00	1	<a href="#">VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers</a>

### 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	<a href="#">MCRR</a>
1	1	<a href="#">MRRC</a>

### 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:U:D:W != 00x0

Decode fields					Instruction Details			Feature
P	U	W	L	Rn	size	imm8		
0	0	1					UNALLOCATED	-

P	U	W	L	Decode fields Rn	size	imm8	Instruction Details	Feature
0	1				0x		UNALLOCATED	-
0	1		0		10		<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">single-precision scalar</a>	-
0	1		0		11	xxxxxxxx0	<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">double-precision scalar</a>	-
0	1		0		11	xxxxxxxx1	<a href="#">FSTMDBX, FSTMIAX</a> — <a href="#">Increment After</a>	-
0	1		1		10		<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">single-precision scalar</a>	-
0	1		1		11	xxxxxxxx0	<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">double-precision scalar</a>	-
0	1		1		11	xxxxxxxx1	<a href="#">FLDM*X (FLDMDBX, FLDMIAX)</a> — <a href="#">Increment After</a>	-
1		0	0		01		<a href="#">VSTR</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
1		0	0		10		<a href="#">VSTR</a> — <a href="#">single-precision scalar</a>	-
1		0	0		11		<a href="#">VSTR</a> — <a href="#">double-precision scalar</a>	-
1		0	1	!= 1111	01		<a href="#">VLDR (immediate)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
1		0	1	!= 1111	10		<a href="#">VLDR (immediate)</a> — <a href="#">single-precision scalar</a>	-
1		0	1	!= 1111	11		<a href="#">VLDR (immediate)</a> — <a href="#">double-precision scalar</a>	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">single-precision scalar</a>	-
1	0	1	0		11	xxxxxxxx0	<a href="#">VSTM, VSTMDB, VSTMIA</a> — <a href="#">double-precision scalar</a>	-
1	0	1	0		11	xxxxxxxx1	<a href="#">FSTMDBX, FSTMIAX</a> — <a href="#">Decrement Before</a>	-
1	0	1	1		10		<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">single-precision scalar</a>	-
1	0	1	1		11	xxxxxxxx0	<a href="#">VLDM, VLDMDB, VLDMIA</a> — <a href="#">double-precision scalar</a>	-
1	0	1	1		11	xxxxxxxx1	<a href="#">FLDM*X (FLDMDBX, FLDMIAX)</a> — <a href="#">Decrement Before</a>	-
1		0	1	1111	01		<a href="#">VLDR (literal)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
1		0	1	1111	10		<a href="#">VLDR (literal)</a> — <a href="#">single-precision scalar</a>	-
1		0	1	1111	11		<a href="#">VLDR (literal)</a> — <a href="#">double-precision scalar</a>	-
1	1	1					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:U:D:W != 00x0

P:U:W	D	L	Decode fields Rn	CRd	cp15	Instruction Details
!= 000				!= 0101	0	UNALLOCATED
!= 000	0	1	1111	0101	0	<a href="#">LDC (literal)</a>
!= 000					1	UNALLOCATED
!= 000	1			0101	0	UNALLOCATED
0x1	0	0		0101	0	<a href="#">STC</a> — <a href="#">post-indexed</a>
0x1	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">post-indexed</a>
010	0	0		0101	0	<a href="#">STC</a> — <a href="#">unindexed</a>
010	0	1	!= 1111	0101	0	<a href="#">LDC (immediate)</a> — <a href="#">unindexed</a>
1x0	0	0		0101	0	<a href="#">STC</a> — <a href="#">offset</a>

Decode fields						Instruction Details
P:U:W	D	L	Rn	CRd	cp15	
1×0	0	1	!= 1111	0101	0	<a href="#">LDC (immediate) — offset</a>
1×1	0	0		0101	0	<a href="#">STC — pre-indexed</a>
1×1	0	1	!= 1111	0101	0	<a href="#">LDC (immediate) — pre-indexed</a>

## 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
11101110								op0								1	op1								1						

Decode fields		Instruction details	Feature
op0	op1		
000	000	UNALLOCATED	-
000	001	<a href="#">VMOV (between general-purpose register and half-precision)</a>	FEAT_FP16
000	010	<a href="#">VMOV (between general-purpose register and single-precision)</a>	-
001	010	UNALLOCATED	-
01×	010	UNALLOCATED	-
10×	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	<a href="#">Floating-point move special register</a>	-
	011	<a href="#">Advanced SIMD 8/16/32-bit element move/duplicate</a>	-
	10×	UNALLOCATED	-
	11×	<a href="#">System register 32-bit move</a>	-

## 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		<a href="#">VMSR</a>
1		<a href="#">VMRS</a>

## 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			Instruction Details
opc1	L	opc2	
0×	0		<a href="#">VMOV (general-purpose register to scalar)</a>
	1		<a href="#">VMOV (scalar to general-purpose register)</a>
1×	0	0×	<a href="#">VDUP (general-purpose register)</a>
1×	0	1×	UNALLOCATED

## 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	<a href="#">MCR</a>
1	<a href="#">MRC</a>

## 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						
11101110								op0																10						op1				0			

Decode fields		Instruction details
op0	op1	
1×11	1	<a href="#">Floating-point data-processing (two registers)</a>
1×11	0	<a href="#">Floating-point move immediate</a>
!= 1×11		<a href="#">Floating-point data-processing (three registers)</a>

### 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	Feature
o1	opc2	size	o3		
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000	01	1	<a href="#">VABS — half-precision scalar</a>	FEAT_FP16
0	000	10	0	<a href="#">VMOV (register) — single-precision scalar</a>	-
0	000	10	1	<a href="#">VABS — single-precision scalar</a>	-
0	000	11	0	<a href="#">VMOV (register) — double-precision scalar</a>	-
0	000	11	1	<a href="#">VABS — double-precision scalar</a>	-
0	001	01	0	<a href="#">VNEG — half-precision scalar</a>	FEAT_FP16
0	001	01	1	<a href="#">VSQRT — half-precision scalar</a>	FEAT_FP16
0	001	10	0	<a href="#">VNEG — single-precision scalar</a>	-
0	001	10	1	<a href="#">VSQRT — single-precision scalar</a>	-
0	001	11	0	<a href="#">VNEG — double-precision scalar</a>	-
0	001	11	1	<a href="#">VSQRT — double-precision scalar</a>	-
0	010	01		UNALLOCATED	-
0	010	10	0	<a href="#">VCVTB — half-precision to single-precision</a>	-
0	010	10	1	<a href="#">VCVTT — half-precision to single-precision</a>	-
0	010	11	0	<a href="#">VCVTB — half-precision to double-precision</a>	-
0	010	11	1	<a href="#">VCVTT — half-precision to double-precision</a>	-
0	011	01	0	<a href="#">VCVTB (BFloat16)</a>	FEAT_AA32BF16
0	011	01	1	<a href="#">VCVTT (BFloat16)</a>	FEAT_AA32BF16



Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	011	10	0	<a href="#">VCVTB</a> — <a href="#">single-precision to half-precision</a>	-
0	011	10	1	<a href="#">VCVTT</a> — <a href="#">single-precision to half-precision</a>	-
0	011	11	0	<a href="#">VCVTB</a> — <a href="#">double-precision to half-precision</a>	-
0	011	11	1	<a href="#">VCVTT</a> — <a href="#">double-precision to half-precision</a>	-
0	100	01	0	<a href="#">VCMP</a>	FEAT_FP16
0	100	01	1	<a href="#">VCMPE</a>	FEAT_FP16
0	100	10	0	<a href="#">VCMP</a>	-
0	100	10	1	<a href="#">VCMPE</a>	-
0	100	11	0	<a href="#">VCMP</a>	-
0	100	11	1	<a href="#">VCMPE</a>	-
0	101	01	0	<a href="#">VCMP</a>	FEAT_FP16
0	101	01	1	<a href="#">VCMPE</a>	FEAT_FP16
0	101	10	0	<a href="#">VCMP</a>	-
0	101	10	1	<a href="#">VCMPE</a>	-
0	101	11	0	<a href="#">VCMP</a>	-
0	101	11	1	<a href="#">VCMPE</a>	-
0	110	01	0	<a href="#">VRINTR</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
0	110	01	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
0	110	10	0	<a href="#">VRINTR</a> — <a href="#">single-precision scalar</a>	-
0	110	10	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">single-precision scalar</a>	-
0	110	11	0	<a href="#">VRINTR</a> — <a href="#">double-precision scalar</a>	-
0	110	11	1	<a href="#">VRINTZ</a> (floating-point) — <a href="#">double-precision scalar</a>	-
0	111	01	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">single-precision scalar</a>	-
0	111	10	1	<a href="#">VCVT</a> (between double-precision and single-precision) — <a href="#">single-precision to double-precision</a>	-
0	111	11	0	<a href="#">VRINTX</a> (floating-point) — <a href="#">double-precision scalar</a>	-
0	111	11	1	<a href="#">VCVT</a> (between double-precision and single-precision) — <a href="#">double-precision to single-precision</a>	-
1	000	01		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">half-precision scalar</a>	FEAT_FP16
1	000	10		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">single-precision scalar</a>	-
1	000	11		<a href="#">VCVT</a> (integer to floating-point, floating-point) — <a href="#">double-precision scalar</a>	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	<a href="#">VJCVT</a>	FEAT_JSCVT
1	01x	01		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-
1	01x	11		<a href="#">VCVT</a> (between floating-point and fixed-point, floating-point)	-
1	100	01	0	<a href="#">VCVTR</a>	FEAT_FP16
1	100	01	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	<a href="#">VCVTR</a>	-
1	100	10	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	100	11	0	<a href="#">VCVTR</a>	-
1	100	11	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	-
1	101	01	0	<a href="#">VCVTR</a>	FEAT_FP16
1	101	01	1	<a href="#">VCVT</a> (floating-point to integer, floating-point)	FEAT_FP16

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
1	101	10	0	<a href="#">VCVTR</a>	-
1	101	10	1	<a href="#">VCVT (floating-point to integer, floating-point)</a>	-
1	101	11	0	<a href="#">VCVTR</a>	-
1	101	11	1	<a href="#">VCVT (floating-point to integer, floating-point)</a>	-
1	11x	01		<a href="#">VCVT (between floating-point and fixed-point, floating-point)</a>	FEAT_FP16
1	11x	10		<a href="#">VCVT (between floating-point and fixed-point, floating-point)</a>	-
1	11x	11		<a href="#">VCVT (between floating-point and fixed-point, floating-point)</a>	-

## 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	Feature
size			
00		UNALLOCATED	-
01		<a href="#">VMOV (immediate)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
10		<a href="#">VMOV (immediate)</a> — <a href="#">single-precision scalar</a>	-
11		<a href="#">VMOV (immediate)</a> — <a href="#">double-precision scalar</a>	-

## 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:D:o1 != 1x11

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
!= 111	00		UNALLOCATED	-
000	01	0	<a href="#">VMLA (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
000	01	1	<a href="#">VMLS (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
000	10	0	<a href="#">VMLA (floating-point)</a> — <a href="#">single-precision scalar</a>	-
000	10	1	<a href="#">VMLS (floating-point)</a> — <a href="#">single-precision scalar</a>	-
000	11	0	<a href="#">VMLA (floating-point)</a> — <a href="#">double-precision scalar</a>	-
000	11	1	<a href="#">VMLS (floating-point)</a> — <a href="#">double-precision scalar</a>	-
001	01	0	<a href="#">VNMLS</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
001	01	1	<a href="#">VNMLA</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
001	10	0	<a href="#">VNMLS</a> — <a href="#">single-precision scalar</a>	-
001	10	1	<a href="#">VNMLA</a> — <a href="#">single-precision scalar</a>	-
001	11	0	<a href="#">VNMLS</a> — <a href="#">double-precision scalar</a>	-
001	11	1	<a href="#">VNMLA</a> — <a href="#">double-precision scalar</a>	-
010	01	0	<a href="#">VMUL (floating-point)</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
010	01	1	<a href="#">VNMUL</a> — <a href="#">half-precision scalar</a>	FEAT_FP16
010	10	0	<a href="#">VMUL (floating-point)</a> — <a href="#">single-precision scalar</a>	-
010	10	1	<a href="#">VNMUL</a> — <a href="#">single-precision scalar</a>	-

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
010	11	0	<a href="#">VMUL (floating-point) — double-precision scalar</a>	-
010	11	1	<a href="#">VNMUL — double-precision scalar</a>	-
011	01	0	<a href="#">VADD (floating-point) — half-precision scalar</a>	FEAT_FP16
011	01	1	<a href="#">VSUB (floating-point) — half-precision scalar</a>	FEAT_FP16
011	10	0	<a href="#">VADD (floating-point) — single-precision scalar</a>	-
011	10	1	<a href="#">VSUB (floating-point) — single-precision scalar</a>	-
011	11	0	<a href="#">VADD (floating-point) — double-precision scalar</a>	-
011	11	1	<a href="#">VSUB (floating-point) — double-precision scalar</a>	-
100	01	0	<a href="#">VDIV — half-precision scalar</a>	FEAT_FP16
100	10	0	<a href="#">VDIV — single-precision scalar</a>	-
100	11	0	<a href="#">VDIV — double-precision scalar</a>	-
101	01	0	<a href="#">VFNMS — half-precision scalar</a>	FEAT_FP16
101	01	1	<a href="#">VFNMA — half-precision scalar</a>	FEAT_FP16
101	10	0	<a href="#">VFNMS — single-precision scalar</a>	-
101	10	1	<a href="#">VFNMA — single-precision scalar</a>	-
101	11	0	<a href="#">VFNMS — double-precision scalar</a>	-
101	11	1	<a href="#">VFNMA — double-precision scalar</a>	-
110	01	0	<a href="#">VFMA — half-precision scalar</a>	FEAT_FP16
110	01	1	<a href="#">VFMS — half-precision scalar</a>	FEAT_FP16
110	10	0	<a href="#">VFMA — single-precision scalar</a>	-
110	10	1	<a href="#">VFMS — single-precision scalar</a>	-
110	11	0	<a href="#">VFMA — double-precision scalar</a>	-
110	11	1	<a href="#">VFMS — double-precision scalar</a>	-

## 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
111111						op0			op1								1	op2	op3			op4			op5						

The following constraints also apply to this encoding: op0<2:1> != 11

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			<a href="#">Advanced SIMD three registers of the same length extension</a>
100		0	!= 00	0	0	<a href="#">Floating-point conditional select</a>
101	00xxxxx	0	!= 00		0	<a href="#">Floating-point minNum/maxNum</a>
101	110000	0	!= 00	1	0	<a href="#">Floating-point extraction and insertion</a>
101	111xxx	0	!= 00	1	0	<a href="#">Floating-point directed convert to integer</a>
10x		0	00			<a href="#">Advanced SIMD and floating-point multiply with accumulate</a>
10x		1	0x			<a href="#">Advanced SIMD and floating-point dot product</a>

## 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	Feature
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	<a href="#">VCADD</a> — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	<a href="#">VCADD</a> — 128-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	1	1	UNALLOCATED	-
00	0x	0	0			UNALLOCATED	-
00	0x	0	1			UNALLOCATED	-
00	00	1	0	0	0	UNALLOCATED	-
00	00	1	0	0	1	UNALLOCATED	-
00	00	1	0	1	0	<a href="#">VMMLA</a>	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	<a href="#">VDOT (vector)</a> — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	<a href="#">VDOT (vector)</a> — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	<a href="#">VFMAL (vector)</a>	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	<a href="#">VSMMLA</a>	FEAT_AA32I8MM
00	10	1	0	1	1	<a href="#">VUMMLA</a>	FEAT_AA32I8MM
00	10	1	1	0	0	<a href="#">VSDOT (vector)</a> — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	<a href="#">VUDOT (vector)</a> — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	<a href="#">VSDOT (vector)</a> — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	<a href="#">VUDOT (vector)</a> — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	<a href="#">VFMAb, VFMAb (BFloat16, vector)</a>	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	<a href="#">VFMSL (vector)</a>	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	<a href="#">VUSMMLA</a>	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	<a href="#">VUSDOT (vector)</a> — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	<a href="#">VUSDOT (vector)</a> — 128-bit SIMD vector	FEAT_AA32I8MM
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	<a href="#">VCMLA</a>	FEAT_FCMA
10	11	0	1			UNALLOCATED	-
10	11	1	0			UNALLOCATED	-
10	11	1	1			UNALLOCATED	-
11	11	0	1			UNALLOCATED	-
11	11	1	0			UNALLOCATED	-
11	11	1	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																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields size	Instruction Details	Feature
01	<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — half-precision scalar	FEAT_FP16
10	<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — single-precision scalar	-
11	<a href="#">VSELEQ, VSELGE, VSELGT, VSELVS</a> — 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 && size != 00

Decode fields size op	Instruction Details	Feature
01 0	<a href="#">VMAXNM</a> — half-precision scalar	FEAT_FP16
01 1	<a href="#">VMINNM</a> — half-precision scalar	FEAT_FP16
10 0	<a href="#">VMAXNM</a> — single-precision scalar	-
10 1	<a href="#">VMINNM</a> — single-precision scalar	-
11 0	<a href="#">VMAXNM</a> — double-precision scalar	-
11 1	<a href="#">VMINNM</a> — 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 && size != 00

Decode fields size op	Instruction Details	Feature
01	UNALLOCATED	-
10 0	<a href="#">VMOVX</a>	FEAT_FP16
10 1	<a href="#">VINS</a>	FEAT_FP16
11	UNALLOCATED	-

## 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	o1	RM		Vd				1	0	!= 00	op	1	M	0			Vm		
size																															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields				Instruction Details	Feature
o1	RM	size	op		
0		!= 00	1	UNALLOCATED	-
0	00	01	0	<a href="#">VRINTA (floating-point) — half-precision scalar</a>	FEAT_FP16
0	00	10	0	<a href="#">VRINTA (floating-point) — single-precision scalar</a>	-
0	00	11	0	<a href="#">VRINTA (floating-point) — double-precision scalar</a>	-
0	01	01	0	<a href="#">VRINTN (floating-point) — half-precision scalar</a>	FEAT_FP16
0	01	10	0	<a href="#">VRINTN (floating-point) — single-precision scalar</a>	-
0	01	11	0	<a href="#">VRINTN (floating-point) — double-precision scalar</a>	-
0	10	01	0	<a href="#">VRINTP (floating-point) — half-precision scalar</a>	FEAT_FP16
0	10	10	0	<a href="#">VRINTP (floating-point) — single-precision scalar</a>	-
0	10	11	0	<a href="#">VRINTP (floating-point) — double-precision scalar</a>	-
0	11	01	0	<a href="#">VRINTM (floating-point) — half-precision scalar</a>	FEAT_FP16
0	11	10	0	<a href="#">VRINTM (floating-point) — single-precision scalar</a>	-
0	11	11	0	<a href="#">VRINTM (floating-point) — double-precision scalar</a>	-
1	00	01		<a href="#">VCVTA (floating-point) — half-precision scalar</a>	FEAT_FP16
1	00	10		<a href="#">VCVTA (floating-point) — single-precision scalar</a>	-
1	00	11		<a href="#">VCVTA (floating-point) — double-precision scalar</a>	-
1	01	01		<a href="#">VCVTN (floating-point) — half-precision scalar</a>	FEAT_FP16
1	01	10		<a href="#">VCVTN (floating-point) — single-precision scalar</a>	-
1	01	11		<a href="#">VCVTN (floating-point) — double-precision scalar</a>	-
1	10	01		<a href="#">VCVTP (floating-point) — half-precision scalar</a>	FEAT_FP16
1	10	10		<a href="#">VCVTP (floating-point) — single-precision scalar</a>	-
1	10	11		<a href="#">VCVTP (floating-point) — double-precision scalar</a>	-
1	11	01		<a href="#">VCVTM (floating-point) — half-precision scalar</a>	FEAT_FP16
1	11	10		<a href="#">VCVTM (floating-point) — single-precision scalar</a>	-
1	11	11		<a href="#">VCVTM (floating-point) — double-precision scalar</a>	-

## 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	Feature
op1	op2	Q	U		
0			0	<a href="#">VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point</a>	FEAT_FCMA
0	00		1	<a href="#">VFMAL (by scalar)</a>	FEAT_FHM
0	01		1	<a href="#">VFMSL (by scalar)</a>	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	<a href="#">VFMAb, VFMAr (BFloat16, by scalar)</a>	FEAT_AA32BF16
1		0	0	<a href="#">VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point</a>	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	<a href="#">VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point</a>	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		Feature
op1	op2	op4	Q	U			
0	00	0			UNALLOCATED		-
0	00	1	0	0	<a href="#">VDOT (by element) — 64-bit SIMD vector</a>		FEAT_AA32BF16
0	00	1		1	UNALLOCATED		-
0	00	1	1	0	<a href="#">VDOT (by element) — 128-bit SIMD vector</a>		FEAT_AA32BF16
0	01	0			UNALLOCATED		-
0	10	0			UNALLOCATED		-
0	10	1	0	0	<a href="#">VSDOT (by element) — 64-bit SIMD vector</a>		FEAT_DotProd
0	10	1	0	1	<a href="#">VUDOT (by element) — 64-bit SIMD vector</a>		FEAT_DotProd
0	10	1	1	0	<a href="#">VSDOT (by element) — 128-bit SIMD vector</a>		FEAT_DotProd
0	10	1	1	1	<a href="#">VUDOT (by element) — 128-bit SIMD vector</a>		FEAT_DotProd
0	11				UNALLOCATED		-
1		0			UNALLOCATED		-
1	00	1	0	0	<a href="#">VUSDOT (by element) — 64-bit SIMD vector</a>		FEAT_AA32I8MM
1	00	1	0	1	<a href="#">VSUDOT (by element) — 64-bit SIMD vector</a>		FEAT_AA32I8MM
1	00	1	1	0	<a href="#">VUSDOT (by element) — 128-bit SIMD vector</a>		FEAT_AA32I8MM
1	00	1	1	1	<a href="#">VSUDOT (by element) — 128-bit SIMD vector</a>		FEAT_AA32I8MM
1	01	1			UNALLOCATED		-
1	1x	1			UNALLOCATED		-

## 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	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB — T1</a>	
00	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB — T1</a>	
01	0	<a href="#">STM, STMIA, STMEA</a>	
01	1	<a href="#">LDM, LDMIA, LDMFD</a>	
10	0	<a href="#">STMDB, STMFD</a>	
10	1	<a href="#">LDMDB, LDMEA</a>	
11	0	<a href="#">SRS, SRSDA, SRSDB, SRSIA, SRSIB — T2</a>	
11	1	<a href="#">RFE, RFEDA, RFEDB, RFEIA, RFEIB — T2</a>	

## 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
1110100							op0		op1	op2			op3																		

The following constraints also apply to this encoding: op0<1> == 1

Decode fields				Instruction details
op0	op1	op2	op3	
0010				<a href="#">Load/store exclusive</a>
0110	0		000	UNALLOCATED
0110	1		000	<a href="#">TBB, TBH</a>
0110			01x	<a href="#">Load/store exclusive byte/half/dual</a>
0110			1xx	<a href="#">Load-acquire / Store-release</a>
0x11		!= 1111		<a href="#">Load/store dual (immediate, post-indexed)</a>
1x10		!= 1111		<a href="#">Load/store dual (immediate)</a>
1x11		!= 1111		<a href="#">Load/store dual (immediate, pre-indexed)</a>
!= 0xx0		1111		<a href="#">LDRD (literal)</a>

### Load/store exclusive

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	0	1	0	L	Rn				Rt				Rd			imm8								

Decode fields		Instruction Details
L		
0		<a href="#">STREX</a>
1		<a href="#">LDREX</a>

### Load/store exclusive byte/half/dual

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			0 1		sz		Rd						

Decode fields			Instruction Details
L	sz		
0	00		<a href="#">STREXB</a>
0	01		<a href="#">STREXH</a>
0	10		UNALLOCATED
0	11		<a href="#">STREXD</a>
1	00		<a href="#">LDREXB</a>
1	01		<a href="#">LDREXH</a>
1	10		UNALLOCATED
1	11		<a href="#">LDREXD</a>

### 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					

Decode fields			Instruction Details
L	op	sz	
0	0	00	<a href="#">STLB</a>
0	0	01	<a href="#">STLH</a>
0	0	10	<a href="#">STL</a>



Decode fields			Instruction Details
L	op	sz	
0	0	11	UNALLOCATED
0	1	00	<a href="#">STLEXB</a>
0	1	01	<a href="#">STLEXH</a>
0	1	10	<a href="#">STLEX</a>
0	1	11	<a href="#">STLEXD</a>
1	0	00	<a href="#">LDAB</a>
1	0	01	<a href="#">LDAH</a>
1	0	10	<a href="#">LDA</a>
1	0	11	UNALLOCATED
1	1	00	<a href="#">LDAEXB</a>
1	1	01	<a href="#">LDAEXH</a>
1	1	10	<a href="#">LDAEX</a>
1	1	11	<a href="#">LDAEXD</a>

### 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							
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
L		
0		<a href="#">STRD (immediate)</a>
1		<a href="#">LDRD (immediate)</a>

### 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							
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
L		
0		<a href="#">STRD (immediate)</a>
1		<a href="#">LDRD (immediate)</a>

### 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										
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields	Instruction Details
<b>L</b>	
0	<a href="#">STRD (immediate)</a>
1	<a href="#">LDRD (immediate)</a>

**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:stype	Rd		
0000	0		!= 0000011		<a href="#">AND, ANDS (register)</a>	— <a href="#">AND</a> , shift or rotate by value
0000	0		0000011		<a href="#">AND, ANDS (register)</a>	— <a href="#">AND</a> , rotate right with extend
0000	1		!= 0000011	!= 1111	<a href="#">AND, ANDS (register)</a>	— <a href="#">ANDS</a> , shift or rotate by value
0000	1		!= 0000011	1111	<a href="#">TST (register)</a>	— shift or rotate by value
0000	1		0000011	!= 1111	<a href="#">AND, ANDS (register)</a>	— <a href="#">ANDS</a> , rotate right with extend
0000	1		0000011	1111	<a href="#">TST (register)</a>	— rotate right with extend
0001			!= 0000011		<a href="#">BIC, BICS (register)</a>	— shift or rotate by value
0001			0000011		<a href="#">BIC, BICS (register)</a>	— rotate right with extend
0010	0	!= 1111	!= 0000011		<a href="#">ORR, ORRS (register)</a>	— <a href="#">ORR</a> , shift or rotate by value
0010	0	!= 1111	0000011		<a href="#">ORR, ORRS (register)</a>	— <a href="#">ORR</a> , rotate right with extend
0010	0	1111	!= 0000011		<a href="#">MOV, MOVS (register)</a>	— <a href="#">MOV</a> , shift or rotate by value
0010	0	1111	0000011		<a href="#">MOV, MOVS (register)</a>	— <a href="#">MOV</a> , rotate right with extend
0010	1	!= 1111	!= 0000011		<a href="#">ORR, ORRS (register)</a>	— <a href="#">ORRS</a> , shift or rotate by value
0010	1	!= 1111	0000011		<a href="#">ORR, ORRS (register)</a>	— <a href="#">ORRS</a> , rotate right with extend
0010	1	1111	!= 0000011		<a href="#">MOV, MOVS (register)</a>	— <a href="#">MOVS</a> , shift or rotate by value
0010	1	1111	0000011		<a href="#">MOV, MOVS (register)</a>	— <a href="#">MOVS</a> , rotate right with extend
0011	0	!= 1111	!= 0000011		<a href="#">ORN, ORNS (register)</a>	— <a href="#">ORN</a> , shift or rotate by value
0011	0	!= 1111	0000011		<a href="#">ORN, ORNS (register)</a>	— <a href="#">ORN</a> , rotate right with extend
0011	0	1111	!= 0000011		<a href="#">MVN, MVNS (register)</a>	— <a href="#">MVN</a> , shift or rotate by value
0011	0	1111	0000011		<a href="#">MVN, MVNS (register)</a>	— <a href="#">MVN</a> , rotate right with extend
0011	1	!= 1111	!= 0000011		<a href="#">ORN, ORNS (register)</a>	— <a href="#">ORNS</a> , shift or rotate by value
0011	1	!= 1111	0000011		<a href="#">ORN, ORNS (register)</a>	— <a href="#">ORNS</a> , rotate right with extend
0011	1	1111	!= 0000011		<a href="#">MVN, MVNS (register)</a>	— <a href="#">MVNS</a> , shift or rotate by value
0011	1	1111	0000011		<a href="#">MVN, MVNS (register)</a>	— <a href="#">MVNS</a> , rotate right with extend
0100	0		!= 0000011		<a href="#">EOR, EORS (register)</a>	— <a href="#">EOR</a> , shift or rotate by value
0100	0		0000011		<a href="#">EOR, EORS (register)</a>	— <a href="#">EOR</a> , rotate right with extend
0100	1		!= 0000011	!= 1111	<a href="#">EOR, EORS (register)</a>	— <a href="#">EORS</a> , shift or rotate by value
0100	1		!= 0000011	1111	<a href="#">TEQ (register)</a>	— shift or rotate by value
0100	1		0000011	!= 1111	<a href="#">EOR, EORS (register)</a>	— <a href="#">EORS</a> , rotate right with extend
0100	1		0000011	1111	<a href="#">TEQ (register)</a>	— rotate right with extend
0101					UNALLOCATED	
0110	0		xxxxx00		<a href="#">PKHBT, PKHTB</a>	— <a href="#">PKHBT</a>
0110	0		xxxxx01		UNALLOCATED	
0110	0		xxxxx10		<a href="#">PKHBT, PKHTB</a>	— <a href="#">PKHTB</a>
0110	0		xxxxx11		UNALLOCATED	
0111					UNALLOCATED	
1000	0	!= 1101	!= 0000011		<a href="#">ADD, ADDS (register)</a>	— <a href="#">ADD</a> , shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:type	Rd	
1000	0	!= 1101	0000011		<a href="#">ADD, ADDS (register)</a> — <a href="#">ADD, rotate right with extend</a>
1000	0	1101	!= 0000011		<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADD, shift or rotate by value</a>
1000	0	1101	0000011		<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADD, rotate right with extend</a>
1000	1		!= 0000011	1111	<a href="#">CMN (register)</a> — <a href="#">shift or rotate by value</a>
1000	1	!= 1101	!= 0000011	!= 1111	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADDS, shift or rotate by value</a>
1000	1	!= 1101	0000011	!= 1111	<a href="#">ADD, ADDS (register)</a> — <a href="#">ADDS, rotate right with extend</a>
1000	1		0000011	1111	<a href="#">CMN (register)</a> — <a href="#">rotate right with extend</a>
1000	1	1101	!= 0000011	!= 1111	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADDS, shift or rotate by value</a>
1000	1	1101	0000011	!= 1111	<a href="#">ADD, ADDS (SP plus register)</a> — <a href="#">ADDS, rotate right with extend</a>
1001					UNALLOCATED
1010			!= 0000011		<a href="#">ADC, ADCS (register)</a> — <a href="#">shift or rotate by value</a>
1010			0000011		<a href="#">ADC, ADCS (register)</a> — <a href="#">rotate right with extend</a>
1011			!= 0000011		<a href="#">SBC, SBCS (register)</a> — <a href="#">shift or rotate by value</a>
1011			0000011		<a href="#">SBC, SBCS (register)</a> — <a href="#">rotate right with extend</a>
1100					UNALLOCATED
1101	0	!= 1101	!= 0000011		<a href="#">SUB, SUBS (register)</a> — <a href="#">SUB, shift or rotate by value</a>
1101	0	!= 1101	0000011		<a href="#">SUB, SUBS (register)</a> — <a href="#">SUB, rotate right with extend</a>
1101	0	1101	!= 0000011		<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUB, shift or rotate by value</a>
1101	0	1101	0000011		<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUB, rotate right with extend</a>
1101	1		!= 0000011	1111	<a href="#">CMP (register)</a> — <a href="#">shift or rotate by value</a>
1101	1	!= 1101	!= 0000011	!= 1111	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUBS, shift or rotate by value</a>
1101	1	!= 1101	0000011	!= 1111	<a href="#">SUB, SUBS (register)</a> — <a href="#">SUBS, rotate right with extend</a>
1101	1		0000011	1111	<a href="#">CMP (register)</a> — <a href="#">rotate right with extend</a>
1101	1	1101	!= 0000011	!= 1111	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUBS, shift or rotate by value</a>
1101	1	1101	0000011	!= 1111	<a href="#">SUB, SUBS (SP minus register)</a> — <a href="#">SUBS, rotate right with extend</a>
1110			!= 0000011		<a href="#">RSB, RSBS (register)</a> — <a href="#">shift or rotate by value</a>
1110			0000011		<a href="#">RSB, RSBS (register)</a> — <a href="#">rotate right with extend</a>
1111					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	
11110					op0	op1					op2	1					op3	op4					op5									

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0		0	<a href="#">MSR (register)</a>
0	1110	0x	0x0		1	<a href="#">MSR (Banked register)</a>
0	1110	10	0x0	000		<a href="#">Hints</a>
0	1110	10	0x0	!= 000		<a href="#">Change processor state</a>
0	1110	11	0x0			<a href="#">Miscellaneous system</a>
0	1111	00	0x0			<a href="#">BXJ</a>
0	1111	01	0x0			<a href="#">Exception return</a>
0	1111	1x	0x0		0	<a href="#">MRS</a>
0	1111	1x	0x0		1	<a href="#">MRS (Banked register)</a>
1	1110	00	000			<a href="#">DCPS</a>

1	1110	00	010			UNALLOCATED
1	1110	01	0x0			UNALLOCATED
1	1110	1x	0x0			UNALLOCATED
1	1111	0x	0x0			UNALLOCATED
1	1111	1x	0x0			<a href="#">Exception generation</a>
	!= 111x		0x0			<a href="#">B — T3</a>
			0x1			<a href="#">B — T4</a>
			1x0			<a href="#">BL, BLX (immediate) — T2</a>
			1x1			<a href="#">BL, BLX (immediate) — T1</a>

## 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		hint					option	

Decode fields		Instruction Details	Feature
hint	option		
0000	0000	<a href="#">NOP</a>	-
0000	0001	<a href="#">YIELD</a>	-
0000	0010	<a href="#">WFE</a>	-
0000	0011	<a href="#">WFI</a>	-
0000	0100	<a href="#">SEV</a>	-
0000	0101	<a href="#">SEVL</a>	-
0000	011x	Reserved hint, behaves as NOP	-
0000	1xxx	Reserved hint, behaves as NOP	-
0001	0000	<a href="#">ESB</a>	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	<a href="#">TSB</a>	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP	-
0001	0100	<a href="#">CSDB</a>	-
0001	0101	Reserved hint, behaves as NOP	-
0001	0110	<a href="#">CLRBHB</a>	FEAT_CLRBHB
0001	0111	Reserved hint, behaves as NOP	-
0001	1xxx	Reserved hint, behaves as NOP	-
001x		Reserved hint, behaves as NOP	-
01xx		Reserved hint, behaves as NOP	-
10xx		Reserved hint, behaves as NOP	-
110x		Reserved hint, behaves as NOP	-
1110		Reserved hint, behaves as NOP	-
1111		<a href="#">DBG</a>	-

## 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:M != 000

Decode fields		Instruction Details
imod	M	
00	1	<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">change mode</a>
01		UNALLOCATED
10		<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt enable and change mode</a>
11		<a href="#">CPS, CPSID, CPSIE</a> — <a href="#">interrupt disable and change mode</a>

## 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	Feature
opc	option		
000x		UNALLOCATED	-
0010		<a href="#">CLREX</a>	-
0011		UNALLOCATED	-
0100	!= 0x00	<a href="#">DSB</a>	-
0100	0000	<a href="#">SSBB</a>	-
0100	0100	<a href="#">PSSBB</a>	-
0101		<a href="#">DMB</a>	-
0110		<a href="#">ISB</a>	-
0111		<a href="#">SB</a>	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		<a href="#">SUB, SUBS (immediate)</a>
111000000000		<a href="#">ERET</a>

## 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	imm10								opt			

Decode fields		opt	Instruction Details
imm4	imm10		
!= 1111			UNALLOCATED
1111	!= 0000000000		UNALLOCATED
1111	0000000000	00	UNALLOCATED
1111	0000000000	01	<a href="#">DCPS1</a>
1111	0000000000	10	<a href="#">DCPS2</a>
1111	0000000000	11	<a href="#">DCPS3</a>

## 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	<a href="#">HVC</a>
0	1	UNALLOCATED
1	0	<a href="#">SMC</a>
1	1	<a href="#">UDF</a>

## 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		Rn	Rd	Instruction Details
op1	S			
0000	0			<a href="#">AND, ANDS (immediate)</a> — <a href="#">AND</a>
0000	1		!= 1111	<a href="#">AND, ANDS (immediate)</a> — <a href="#">ANDS</a>
0000	1		1111	<a href="#">TST (immediate)</a>
0001				<a href="#">BIC, BICS (immediate)</a>
0010	0	!= 1111		<a href="#">ORR, ORRS (immediate)</a> — <a href="#">ORR</a>
0010	0	1111		<a href="#">MOV, MOVS (immediate)</a> — <a href="#">MOV</a>
0010	1	!= 1111		<a href="#">ORR, ORRS (immediate)</a> — <a href="#">ORRS</a>
0010	1	1111		<a href="#">MOV, MOVS (immediate)</a> — <a href="#">MOVS</a>
0011	0	!= 1111		<a href="#">ORN, ORNS (immediate)</a> — <a href="#">not flag setting</a>
0011	0	1111		<a href="#">MVN, MVNS (immediate)</a> — <a href="#">MVN</a>
0011	1	!= 1111		<a href="#">ORN, ORNS (immediate)</a> — <a href="#">flag setting</a>
0011	1	1111		<a href="#">MVN, MVNS (immediate)</a> — <a href="#">MVNS</a>
0100	0			<a href="#">EOR, EORS (immediate)</a> — <a href="#">EOR</a>
0100	1		!= 1111	<a href="#">EOR, EORS (immediate)</a> — <a href="#">EORS</a>
0100	1		1111	<a href="#">TEQ (immediate)</a>
0101				UNALLOCATED
011x				UNALLOCATED
1000	0	!= 1101		<a href="#">ADD, ADDS (immediate)</a> — <a href="#">ADD</a>
1000	0	1101		<a href="#">ADD, ADDS (SP plus immediate)</a> — <a href="#">ADD</a>
1000	1	!= 1101	!= 1111	<a href="#">ADD, ADDS (immediate)</a> — <a href="#">ADDS</a>
1000	1	1101	!= 1111	<a href="#">ADD, ADDS (SP plus immediate)</a> — <a href="#">ADDS</a>
1000	1		1111	<a href="#">CMN (immediate)</a>
1001				UNALLOCATED
1010				<a href="#">ADC, ADCS (immediate)</a>
1011				<a href="#">SBC, SBCS (immediate)</a>
1100				UNALLOCATED
1101	0	!= 1101		<a href="#">SUB, SUBS (immediate)</a> — <a href="#">SUB</a>
1101	0	1101		<a href="#">SUB, SUBS (SP minus immediate)</a> — <a href="#">SUB</a>
1101	1	!= 1101	!= 1111	<a href="#">SUB, SUBS (immediate)</a> — <a href="#">SUBS</a>
1101	1	1101	!= 1111	<a href="#">SUB, SUBS (SP minus immediate)</a> — <a href="#">SUBS</a>

Decode fields				Instruction Details
op1	S	Rn	Rd	
1101	1		1111	<a href="#">CMP (immediate)</a>
1110				<a href="#">RSB, RSBS (immediate)</a>
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	<a href="#">Data-processing (simple immediate)</a>
0	10	<a href="#">Move Wide (16-bit immediate)</a>
0	11	UNALLOCATED
1		<a href="#">Saturate, Bitfield</a>

## 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	!= 11x1	<a href="#">ADD, ADDS (immediate)</a>
0	0	1101	<a href="#">ADD, ADDS (SP plus immediate)</a>
0	0	1111	<a href="#">ADR — T3</a>
0	1		UNALLOCATED
1	0		UNALLOCATED
1	1	!= 11x1	<a href="#">SUB, SUBS (immediate)</a>
1	1	1101	<a href="#">SUB, SUBS (SP minus immediate)</a>
1	1	1111	<a href="#">ADR — T2</a>

## 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		<a href="#">MOV, MOVS (immediate)</a>
1		<a href="#">MOVT</a>

## 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			<a href="#">SSAT</a> — <a href="#">logical shift left</a>
001		!= 00000	<a href="#">SSAT</a> — <a href="#">arithmetic shift right</a>
001		00000	<a href="#">SSAT16</a>
010			<a href="#">SBFX</a>
011	!= 1111		<a href="#">BFI</a>
011	1111		<a href="#">BFC</a>
100			<a href="#">USAT</a> — <a href="#">logical shift left</a>
101		!= 00000	<a href="#">USAT</a> — <a href="#">arithmetic shift right</a>
101		00000	<a href="#">USAT16</a>
110			<a href="#">UBFX</a>
111			UNALLOCATED

## 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
11111001								op0		0										op1											

Decode fields		Instruction details
op0	op1	
0		<a href="#">Advanced SIMD load/store multiple structures</a>
1	11	<a href="#">Advanced SIMD load single structure to all lanes</a>
1	!= 11	<a href="#">Advanced SIMD load/store single structure to one lane</a>

## 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						

Decode fields			Instruction Details
L	itype	Rm	
0	000x	!= 11x1	<a href="#">VST4 (multiple 4-element structures)</a>
0	000x	1101	<a href="#">VST4 (multiple 4-element structures)</a>
0	000x	1111	<a href="#">VST4 (multiple 4-element structures)</a>
0	0010	!= 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0010	1101	<a href="#">VST1 (multiple single elements)</a>
0	0010	1111	<a href="#">VST1 (multiple single elements)</a>
0	0011	!= 11x1	<a href="#">VST2 (multiple 2-element structures)</a>
0	0011	1101	<a href="#">VST2 (multiple 2-element structures)</a>
0	0011	1111	<a href="#">VST2 (multiple 2-element structures)</a>
0	010x	!= 11x1	<a href="#">VST3 (multiple 3-element structures)</a>
0	010x	1101	<a href="#">VST3 (multiple 3-element structures)</a>
0	010x	1111	<a href="#">VST3 (multiple 3-element structures)</a>
0	0110	!= 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0110	1101	<a href="#">VST1 (multiple single elements)</a>
0	0110	1111	<a href="#">VST1 (multiple single elements)</a>



Decode fields			Instruction Details
L	itype	Rm	
0	0111	!= 11x1	<a href="#">VST1 (multiple single elements)</a>
0	0111	1101	<a href="#">VST1 (multiple single elements)</a>
0	0111	1111	<a href="#">VST1 (multiple single elements)</a>
0	100x	!= 11x1	<a href="#">VST2 (multiple 2-element structures)</a>
0	100x	1101	<a href="#">VST2 (multiple 2-element structures)</a>
0	100x	1111	<a href="#">VST2 (multiple 2-element structures)</a>
0	1010	!= 11x1	<a href="#">VST1 (multiple single elements)</a>
0	1010	1101	<a href="#">VST1 (multiple single elements)</a>
0	1010	1111	<a href="#">VST1 (multiple single elements)</a>
1	000x	!= 11x1	<a href="#">VLD4 (multiple 4-element structures)</a>
1	000x	1101	<a href="#">VLD4 (multiple 4-element structures)</a>
1	000x	1111	<a href="#">VLD4 (multiple 4-element structures)</a>
1	0010	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0010	1101	<a href="#">VLD1 (multiple single elements)</a>
1	0010	1111	<a href="#">VLD1 (multiple single elements)</a>
1	0011	!= 11x1	<a href="#">VLD2 (multiple 2-element structures)</a>
1	0011	1101	<a href="#">VLD2 (multiple 2-element structures)</a>
1	0011	1111	<a href="#">VLD2 (multiple 2-element structures)</a>
1	010x	!= 11x1	<a href="#">VLD3 (multiple 3-element structures)</a>
1	010x	1101	<a href="#">VLD3 (multiple 3-element structures)</a>
1	010x	1111	<a href="#">VLD3 (multiple 3-element structures)</a>
	1011		UNALLOCATED
1	0110	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0110	1101	<a href="#">VLD1 (multiple single elements)</a>
1	0110	1111	<a href="#">VLD1 (multiple single elements)</a>
1	0111	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	0111	1101	<a href="#">VLD1 (multiple single elements)</a>
1	0111	1111	<a href="#">VLD1 (multiple single elements)</a>
	11xx		UNALLOCATED
1	100x	!= 11x1	<a href="#">VLD2 (multiple 2-element structures)</a>
1	100x	1101	<a href="#">VLD2 (multiple 2-element structures)</a>
1	100x	1111	<a href="#">VLD2 (multiple 2-element structures)</a>
1	1010	!= 11x1	<a href="#">VLD1 (multiple single elements)</a>
1	1010	1101	<a href="#">VLD1 (multiple single elements)</a>
1	1010	1111	<a href="#">VLD1 (multiple single elements)</a>

### 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		!= 11x1	<a href="#">VLD1 (single element to all lanes)</a>
1	00		1101	<a href="#">VLD1 (single element to all lanes)</a>
1	00		1111	<a href="#">VLD1 (single element to all lanes)</a>

Decode fields				Instruction Details
L	N	a	Rm	
1	01		!= 11x1	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	01		1101	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	01		1111	<a href="#">VLD2 (single 2-element structure to all lanes)</a>
1	10	0	!= 11x1	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	0	1101	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	0	1111	<a href="#">VLD3 (single 3-element structure to all lanes)</a>
1	10	1		UNALLOCATED
1	11		!= 11x1	<a href="#">VLD4 (single 4-element structure to all lanes)</a>
1	11		1101	<a href="#">VLD4 (single 4-element structure to all lanes)</a>
1	11		1111	<a href="#">VLD4 (single 4-element structure to all lanes)</a>

### 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 && size != 11

Decode fields				Instruction Details
L	size	N	Rm	
0	00	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	00	00	1101	<a href="#">VST1 (single element from one lane)</a>
0	00	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	00	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	00	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	00	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	00	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	00	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	01	00	1101	<a href="#">VST1 (single element from one lane)</a>
0	01	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	01	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	01	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	01	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	01	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	00	!= 11x1	<a href="#">VST1 (single element from one lane)</a>
0	10	00	1101	<a href="#">VST1 (single element from one lane)</a>

Decode fields				Instruction Details
L	size	N	Rm	
0	10	00	1111	<a href="#">VST1 (single element from one lane)</a>
0	10	01	!= 11x1	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	01	1101	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	01	1111	<a href="#">VST2 (single 2-element structure from one lane)</a>
0	10	10	!= 11x1	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	10	1101	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	10	1111	<a href="#">VST3 (single 3-element structure from one lane)</a>
0	10	11	!= 11x1	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	11	1101	<a href="#">VST4 (single 4-element structure from one lane)</a>
0	10	11	1111	<a href="#">VST4 (single 4-element structure from one lane)</a>
1	00	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	00	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	00	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	00	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	00	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	00	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	00	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	00	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	01	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	01	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	01	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	01	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	01	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	01	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	00	!= 11x1	<a href="#">VLD1 (single element to one lane)</a>
1	10	00	1101	<a href="#">VLD1 (single element to one lane)</a>
1	10	00	1111	<a href="#">VLD1 (single element to one lane)</a>
1	10	01	!= 11x1	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	01	1101	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	01	1111	<a href="#">VLD2 (single 2-element structure to one lane)</a>
1	10	10	!= 11x1	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	10	1101	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	10	1111	<a href="#">VLD3 (single 3-element structure to one lane)</a>
1	10	11	!= 11x1	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	11	1101	<a href="#">VLD4 (single 4-element structure to one lane)</a>
1	10	11	1111	<a href="#">VLD4 (single 4-element structure to one lane)</a>

## 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
1111100								op0			op1	op2				op3															

The following constraints also apply to this encoding: op0<1>:op1 != 10

Decode fields				Instruction details
op0	op1	op2	op3	
00		!= 1111	000000	<a href="#">Load/store, unsigned (register offset)</a>
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	<a href="#">Load/store, unsigned (immediate, post-indexed)</a>
00		!= 1111	1100xx	<a href="#">Load/store, unsigned (negative immediate)</a>
00		!= 1111	1110xx	<a href="#">Load/store, unsigned (unprivileged)</a>
00		!= 1111	11x1xx	<a href="#">Load/store, unsigned (immediate, pre-indexed)</a>
01		!= 1111		<a href="#">Load/store, unsigned (positive immediate)</a>
0x		1111		<a href="#">Load, unsigned (literal)</a>
10	1	!= 1111	000000	<a href="#">Load/store, signed (register offset)</a>
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	<a href="#">Load/store, signed (immediate, post-indexed)</a>
10	1	!= 1111	1100xx	<a href="#">Load/store, signed (negative immediate)</a>
10	1	!= 1111	1110xx	<a href="#">Load/store, signed (unprivileged)</a>
10	1	!= 1111	11x1xx	<a href="#">Load/store, signed (immediate, pre-indexed)</a>
11	1	!= 1111		<a href="#">Load/store, signed (positive immediate)</a>
1x	1	1111		<a href="#">Load, signed (literal)</a>

## 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 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		<a href="#">STRB (register)</a>
00	1	!= 1111	<a href="#">LDRB (register)</a>
00	1	1111	<a href="#">PLD, PLDW (register) — preload read</a>

Decode fields			Instruction Details
size	L	Rt	
01	0		<a href="#">STRH (register)</a>
01	1	!= 1111	<a href="#">LDRH (register)</a>
01	1	1111	<a href="#">PLD, PLDW (register)</a> — <a href="#">preload write</a>
10	0		<a href="#">STR (register)</a>
10	1		<a href="#">LDR (register)</a>
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		<a href="#">STRB (immediate)</a>
00	1		<a href="#">LDRB (immediate)</a>
01	0		<a href="#">STRH (immediate)</a>
01	1		<a href="#">LDRH (immediate)</a>
10	0		<a href="#">STR (immediate)</a>
10	1		<a href="#">LDR (immediate)</a>
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	0		<a href="#">STRB (immediate)</a>
00	1	!= 1111	<a href="#">LDRB (immediate)</a>
00	1	1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload read</a>
01	0		<a href="#">STRH (immediate)</a>
01	1	!= 1111	<a href="#">LDRH (immediate)</a>
01	1	1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload write</a>
10	0		<a href="#">STR (immediate)</a>
10	1		<a href="#">LDR (immediate)</a>
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 && Rn != 1111

Decode fields size	L	Instruction Details
00	0	<a href="#">STRBT</a>
00	1	<a href="#">LDRBT</a>
01	0	<a href="#">STRHT</a>
01	1	<a href="#">LDRHT</a>
10	0	<a href="#">STRT</a>
10	1	<a href="#">LDRT</a>
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	L	Instruction Details
00	0	<a href="#">STRB (immediate)</a>
00	1	<a href="#">LDRB (immediate)</a>
01	0	<a href="#">STRH (immediate)</a>
01	1	<a href="#">LDRH (immediate)</a>
10	0	<a href="#">STR (immediate)</a>
10	1	<a href="#">LDR (immediate)</a>
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 && Rn != 1111

Decode fields size	L	Rt	Instruction Details
00	0		<a href="#">STRB (immediate)</a>
00	1	!= 1111	<a href="#">LDRB (immediate)</a>
00	1	1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload read</a>

Decode fields			Instruction Details
size	L	Rt	
01	0		<a href="#">STRH (immediate)</a>
01	1	!= 1111	<a href="#">LDRH (immediate)</a>
01	1	1111	<a href="#">PLD, PLDW (immediate)</a> — <a href="#">preload write</a>
10	0		<a href="#">STR (immediate)</a>
10	1		<a href="#">LDR (immediate)</a>

### 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	
0x	1	1111	<a href="#">PLD (literal)</a>
00	1	!= 1111	<a href="#">LDRB (literal)</a>
01	1	!= 1111	<a href="#">LDRH (literal)</a>
10	1		<a href="#">LDR (literal)</a>
11			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 && Rn != 1111

Decode fields			Instruction Details
size	L	Rt	
00	!= 1111		<a href="#">LDRSB (register)</a>
00		1111	<a href="#">PLI (register)</a>
01	!= 1111		<a href="#">LDRSH (register)</a>
01		1111	Reserved hint, behaves as NOP
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00		<a href="#">LDRSB (immediate)</a>

Decode fields size	Instruction Details
01	<a href="#">LDRSH (immediate)</a>
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Rt	Instruction Details
00	!= 1111	<a href="#">LDRSB (immediate)</a>
00	1111	<a href="#">PLI (immediate, literal)</a>
01	!= 1111	<a href="#">LDRSH (immediate)</a>
01	1111	Reserved hint, behaves as NOP
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 && Rn != 1111

Decode fields size	Instruction Details
00	<a href="#">LDRSBT</a>
01	<a href="#">LDRSHT</a>
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																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields size	Instruction Details
00	<a href="#">LDRSB (immediate)</a>
01	<a href="#">LDRSH (immediate)</a>
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				imm12												
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	Rt	
00	!= 1111	<a href="#">LDRSB (immediate)</a>
00	1111	<a href="#">PLI (immediate, literal)</a>
01	!= 1111	<a href="#">LDRSH (immediate)</a>
01	1111	Reserved hint, behaves as NOP

## 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	<a href="#">LDRSB (literal)</a>
00	1111	<a href="#">PLI (immediate, literal)</a>
01	!= 1111	<a href="#">LDRSH (literal)</a>
01	1111	Reserved hint, behaves as NOP
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
11111010								op0								op1				op2											

Decode fields			Instruction details
op0	op1	op2	
0	1111	0000	<a href="#">MOV, MOVS (register-shifted register)</a> — <a href="#">T2, Flag setting</a>
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	<a href="#">Register extends</a>
1	1111	0xxx	<a href="#">Parallel add-subtract</a>
1	1111	10xx	<a href="#">Data-processing (two source registers)</a>
1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

## 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	<a href="#">SXTAH</a>
00	0	1111	<a href="#">SXTH</a>
00	1	!= 1111	<a href="#">UXTAH</a>
00	1	1111	<a href="#">UXTH</a>
01	0	!= 1111	<a href="#">SXTAB16</a>
01	0	1111	<a href="#">SXTB16</a>
01	1	!= 1111	<a href="#">UXTAB16</a>
01	1	1111	<a href="#">UXTB16</a>
10	0	!= 1111	<a href="#">SXTAB</a>
10	0	1111	<a href="#">SXTB</a>
10	1	!= 1111	<a href="#">UXTAB</a>
10	1	1111	<a href="#">UXTB</a>
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	
000	0	0	0	<a href="#">SADD8</a>
000	0	0	1	<a href="#">QADD8</a>
000	0	1	0	<a href="#">SHADD8</a>
000	0	1	1	UNALLOCATED
000	1	0	0	<a href="#">UADD8</a>
000	1	0	1	<a href="#">UQADD8</a>
000	1	1	0	<a href="#">UHADD8</a>
000	1	1	1	UNALLOCATED
001	0	0	0	<a href="#">SADD16</a>
001	0	0	1	<a href="#">QADD16</a>
001	0	1	0	<a href="#">SHADD16</a>
001	0	1	1	UNALLOCATED
001	1	0	0	<a href="#">UADD16</a>
001	1	0	1	<a href="#">UQADD16</a>
001	1	1	0	<a href="#">UHADD16</a>
001	1	1	1	UNALLOCATED
010	0	0	0	<a href="#">SASX</a>
010	0	0	1	<a href="#">QASX</a>
010	0	1	0	<a href="#">SHASX</a>
010	0	1	1	UNALLOCATED
010	1	0	0	<a href="#">UASX</a>
010	1	0	1	<a href="#">UQASX</a>
010	1	1	0	<a href="#">UHASX</a>
010	1	1	1	UNALLOCATED

Decode fields				Instruction Details
op1	U	H	S	
100	0	0	0	<a href="#">SSUB8</a>
100	0	0	1	<a href="#">QSUB8</a>
100	0	1	0	<a href="#">SHSUB8</a>
100	0	1	1	UNALLOCATED
100	1	0	0	<a href="#">USUB8</a>
100	1	0	1	<a href="#">UQSUB8</a>
100	1	1	0	<a href="#">UHSUB8</a>
100	1	1	1	UNALLOCATED
101	0	0	0	<a href="#">SSUB16</a>
101	0	0	1	<a href="#">QSUB16</a>
101	0	1	0	<a href="#">SHSUB16</a>
101	0	1	1	UNALLOCATED
101	1	0	0	<a href="#">USUB16</a>
101	1	0	1	<a href="#">UQSUB16</a>
101	1	1	0	<a href="#">UHSUB16</a>
101	1	1	1	UNALLOCATED
110	0	0	0	<a href="#">SSAX</a>
110	0	0	1	<a href="#">QSAX</a>
110	0	1	0	<a href="#">SHSAX</a>
110	0	1	1	UNALLOCATED
110	1	0	0	<a href="#">USAX</a>
110	1	0	1	<a href="#">UQSAX</a>
110	1	1	0	<a href="#">UHSAX</a>
110	1	1	1	UNALLOCATED
111				UNALLOCATED

### 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	Feature
op1	op2		
000	00	<a href="#">QADD</a>	-
000	01	<a href="#">QDADD</a>	-
000	10	<a href="#">QSUB</a>	-
000	11	<a href="#">QDSUB</a>	-
001	00	<a href="#">REV</a>	-
001	01	<a href="#">REV16</a>	-
001	10	<a href="#">RBIT</a>	-
001	11	<a href="#">REVSH</a>	-
010	00	<a href="#">SEL</a>	-
010	01	UNALLOCATED	-
010	1x	UNALLOCATED	-
011	00	<a href="#">CLZ</a>	-
011	01	UNALLOCATED	-
011	1x	UNALLOCATED	-

Decode fields		Instruction Details	Feature
op1	op2		
100	00	<a href="#">CRC32</a> — <a href="#">CRC32B</a>	FEAT_CRC32
100	01	<a href="#">CRC32</a> — <a href="#">CRC32H</a>	FEAT_CRC32
100	10	<a href="#">CRC32</a> — <a href="#">CRC32W</a>	FEAT_CRC32
100	11	CONSTRAINED UNPREDICTABLE	-
101	00	<a href="#">CRC32C</a> — <a href="#">CRC32CB</a>	FEAT_CRC32
101	01	<a href="#">CRC32C</a> — <a href="#">CRC32CH</a>	FEAT_CRC32
101	10	<a href="#">CRC32C</a> — <a href="#">CRC32CW</a>	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
111110110																				op0											

Decode fields		Instruction details
op0		
00		<a href="#">Multiply and absolute difference</a>
01		UNALLOCATED
1x		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	op1			Rn			Ra			Rd			0 0		op2		Rm						

Decode fields		op2	Instruction Details
op1	Ra		
000	!= 1111	00	<a href="#">MLA, MLAS</a>
000		01	<a href="#">MLS</a>
000		1x	UNALLOCATED
000	1111	00	<a href="#">MUL, MULS</a>
001	!= 1111	00	<a href="#">SMLABB, SMLABT, SMLATB, SMLATT</a> — <a href="#">SMLABB</a>
001	!= 1111	01	<a href="#">SMLABB, SMLABT, SMLATB, SMLATT</a> — <a href="#">SMLABT</a>
001	!= 1111	10	<a href="#">SMLABB, SMLABT, SMLATB, SMLATT</a> — <a href="#">SMLATB</a>
001	!= 1111	11	<a href="#">SMLABB, SMLABT, SMLATB, SMLATT</a> — <a href="#">SMLATT</a>
001	1111	00	<a href="#">SMULBB, SMULBT, SMULTB, SMULTT</a> — <a href="#">SMULBB</a>
001	1111	01	<a href="#">SMULBB, SMULBT, SMULTB, SMULTT</a> — <a href="#">SMULBT</a>
001	1111	10	<a href="#">SMULBB, SMULBT, SMULTB, SMULTT</a> — <a href="#">SMULTB</a>
001	1111	11	<a href="#">SMULBB, SMULBT, SMULTB, SMULTT</a> — <a href="#">SMULTT</a>
010	!= 1111	00	<a href="#">SMLAD, SMLADX</a> — <a href="#">SMLAD</a>
010	!= 1111	01	<a href="#">SMLAD, SMLADX</a> — <a href="#">SMLADX</a>
010		1x	UNALLOCATED

Decode fields		Instruction Details	
op1	Ra	op2	
010	1111	00	<a href="#">SMUAD, SMUADX</a> — <a href="#">SMUAD</a>
010	1111	01	<a href="#">SMUAD, SMUADX</a> — <a href="#">SMUADX</a>
011	!= 1111	00	<a href="#">SMLAWB, SMLAWT</a> — <a href="#">SMLAWB</a>
011	!= 1111	01	<a href="#">SMLAWB, SMLAWT</a> — <a href="#">SMLAWT</a>
011		1x	UNALLOCATED
011	1111	00	<a href="#">SMULWB, SMULWT</a> — <a href="#">SMULWB</a>
011	1111	01	<a href="#">SMULWB, SMULWT</a> — <a href="#">SMULWT</a>
100	!= 1111	00	<a href="#">SMLSD, SMLSDX</a> — <a href="#">SMLSD</a>
100	!= 1111	01	<a href="#">SMLSD, SMLSDX</a> — <a href="#">SMLSDX</a>
100		1x	UNALLOCATED
100	1111	00	<a href="#">SMUSD, SMUSDX</a> — <a href="#">SMUSD</a>
100	1111	01	<a href="#">SMUSD, SMUSDX</a> — <a href="#">SMUSDX</a>
101	!= 1111	00	<a href="#">SMMLA, SMMLAR</a> — <a href="#">SMMLA</a>
101	!= 1111	01	<a href="#">SMMLA, SMMLAR</a> — <a href="#">SMMLAR</a>
101		1x	UNALLOCATED
101	1111	00	<a href="#">SMMUL, SMMULR</a> — <a href="#">SMMUL</a>
101	1111	01	<a href="#">SMMUL, SMMULR</a> — <a href="#">SMMULR</a>
110		00	<a href="#">SMMLS, SMMLSR</a> — <a href="#">SMMLS</a>
110		01	<a href="#">SMMLS, SMMLSR</a> — <a href="#">SMMLSR</a>
110		1x	UNALLOCATED
111	!= 1111	00	<a href="#">USADA8</a>
111		01	UNALLOCATED
111		1x	UNALLOCATED
111	1111	00	<a href="#">USAD8</a>

## 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			Rn			RdLo			RdHi			op2			Rm							

Decode fields		Instruction Details	
op1	op2		
000	!= 0000	UNALLOCATED	
000	0000	<a href="#">SMULL, SMULLS</a>	
001	!= 1111	UNALLOCATED	
001	1111	<a href="#">SDIV</a>	
010	!= 0000	UNALLOCATED	
010	0000	<a href="#">UMULL, UMULLS</a>	
011	!= 1111	UNALLOCATED	
011	1111	<a href="#">UDIV</a>	
100	0000	<a href="#">SMLAL, SMLALS</a>	
100	0001	UNALLOCATED	
100	001x	UNALLOCATED	
100	01xx	UNALLOCATED	
100	1000	<a href="#">SMLALBB, SMLALBT, SMLALTB, SMLALTT</a> — <a href="#">SMLALBB</a>	
100	1001	<a href="#">SMLALBB, SMLALBT, SMLALTB, SMLALTT</a> — <a href="#">SMLALBT</a>	
100	1010	<a href="#">SMLALBB, SMLALBT, SMLALTB, SMLALTT</a> — <a href="#">SMLALTB</a>	

Decode fields		Instruction Details
op1	op2	
100	1011	<a href="#">SMLALBB, SMLALBT, SMLALTB, SMLALTT</a> — <a href="#">SMLALTT</a>
100	1100	<a href="#">SMLALD, SMLALDX</a> — <a href="#">SMLALD</a>
100	1101	<a href="#">SMLALD, SMLALDX</a> — <a href="#">SMLALDX</a>
100	111x	UNALLOCATED
101	0xxx	UNALLOCATED
101	10xx	UNALLOCATED
101	1100	<a href="#">SMLS LD, SMLS LD X</a> — <a href="#">SMLS LD</a>
101	1101	<a href="#">SMLS LD, SMLS LD X</a> — <a href="#">SMLS LD X</a>
101	111x	UNALLOCATED
110	0000	<a href="#">UMLAL, UMLALS</a>
110	0001	UNALLOCATED
110	001x	UNALLOCATED
110	010x	UNALLOCATED
110	0110	<a href="#">UMAAL</a>
110	0111	UNALLOCATED
110	1xxx	UNALLOCATED
111		UNALLOCATED

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54  
Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

# Shared Pseudocode Functions

This page displays common pseudocode functions shared by many pages

## Pseudocodes

### Library pseudocode for aarch32/at/AArch32.AT

```
// AArch32.AT()
// =====
// Perform address translation as per AT instructions.

AArch32.AT(bits(32) vaddress, TranslationStage stage_in, bits(2) el, ATAccess ataccess)
    TranslationStage stage = stage_in;
    SecurityState ss;
    Regime regime;
    boolean eae;

    // ATS1Hx instructions
    if el == EL2 then
        regime = Regime\_EL2;
        eae = TRUE;
        ss = SS\_NonSecure;

    // ATS1Cxx instructions
    elseif 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;

    AddressDescriptor addrdesc;
    SDFTYPE sdftype;
    constant boolean aligned = TRUE;
    bit supersection = '0';

    accdesc = CreateAccDescAT(ss, el, ataccess);

    // Prepare fault fields in case a fault is detected
    fault = NoFault(accdesc, ZeroExtend(vaddress, 64));

    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';

    // ATS12NSOxx instructions
    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);

    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 = bits(64) UNKNOWN;
            AArch32.Abort(fault);

    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);
    return;
```



## Library pseudocode for aarch32/at/AArch32.EncodePARLD

```
// AArch32.EncodePARLD()
// =====
// Returns 64-bit format PAR on address translation instruction.

AArch32.EncodePARLD(AddressDescriptor addrdesc, SecurityState ss)

    if !IsFault(addrdesc) then
        bit ns;
        if ss == SS\_NonSecure then
            ns = bit UNKNOWN;
        elsif addrdesc.paddress.paspace == PAS\_Secure then
            ns = '0';
        else
            ns = '1';
        PAR.F      = '0';
        PAR.SH     = ReportedPARShareability(PAREncodeShareability(addrdesc.memattrs));
        PAR.NS     = ns;
        PAR<10>    = bit IMPLEMENTATION_DEFINED "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> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";         // IMPDEF
    return;
```

## Library pseudocode for aarch32/at/AArch32.EncodePARSD

```
// AArch32.EncodePARSD()
// =====
// Returns 32-bit format PAR on address translation instruction.

AArch32.EncodePARSD(AddressDescriptor addrdesc_in, bit supersection, SecurityState ss)
AddressDescriptor addrdesc = 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;
    bit ns;
    if ss == SS\_NonSecure then
        ns = bit UNKNOWN;
    elsif addrdesc.paddress.paspace == PAS\_Secure then
        ns = '0';
    else
        ns = '1';
    constant bits(2) sh = (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>     = bit IMPLEMENTATION_DEFINED "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> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";       // IMPDEF
return;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusLD

```
// AArch32.PARFaultStatusLD()
// =====
// Fault status field decoding of 64-bit PAR

bits(6) AArch32.PARFaultStatusLD(FaultRecord fault)
    bits(6) syndrome;

    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);
    return syndrome;
```

## Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
    bits(6) syndrome;

    syndrome<5> = if IsExternalAbort(fault) then fault.extflag else '0';
    syndrome<4:0> = EncodeSDFSC(fault.statuscode, fault.level);
    return syndrome;
```

## Library pseudocode for aarch32/at/AArch32.PARInnerAttrs

```
// AArch32.PARInnerAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Inner field.

bits(3) AArch32.PARInnerAttrs(MemoryAttributes memattrs)
    bits(3) result;

    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
    else
        constant MemAttrHints inner = 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
    return result;
```

## Library pseudocode for aarch32/at/AArch32.PAROuterAttrs

```
// AArch32.PAROuterAttrs()
// =====
// Convert orthogonal attributes and hints to 32-bit PAR Outer field.

bits(2) AArch32.PAROuterAttrs(MemoryAttributes memattrs)
    bits(2) result;

    if memattrs.memtype == MemType\_Device then
        result = bits(2) UNKNOWN;
    else
        constant MemAttrHints outer = 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
    return result;
```

### 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.

bits(3) AArch32.ReportedInnerAttrs(bits(3) attrs);
```

### 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.

bits(2) AArch32.ReportedOuterAttrs(bits(2) attrs);
```



```

// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    CacheRecord cache;

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, cache.waynum, cache.level) = DecodeSW(ZeroExtend(regval, 64),
        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;
        CACHE\_OP(cache);
        return;

if EL2Enabled() then
    if PSTATE.EL IN {EL0, EL1} then
        cache.is_vmid_valid = TRUE;
        cache.vmid = VMID[];
    else
        cache.is_vmid_valid = FALSE;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

need_translate = DCInstNeedsTranslation(opscope);
vaddress = regval;

integer size = 0; // by default no watchpoint address
if cacheop == CacheOp\_Invalidate then
    size = DataCacheWatchpointSize();
    vaddress = Align(regval, size);

cache.translated = need_translate;
cache.vaddress = ZeroExtend(vaddress, 64);

if need_translate then
    constant boolean aligned = TRUE;
    constant AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch32.TranslateAddress(vaddress, accdesc,
        aligned, size);

    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = ZeroExtend(regval, 64);
        AArch32.Abort(memaddrdesc.fault);

    cache.paddress = memaddrdesc.paddress;
    if opscope == CacheOpScope\_PoC then
        cache.shareability = memaddrdesc.memattrs.shareability;
    else
        cache.shareability = Shareability\_NSH;
else
    cache.shareability = Shareability UNKNOWN;
    cache.paddress = FullAddress UNKNOWN;

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;

    CACHE\_OP(cache);
    return;

```

## Library pseudocode for aarch32/debug/VCRMatch/AArch32.VCRMatch

```

// AArch32.VCRMatch()
// =====

boolean AArch32.VCRMatch(bits(32) vaddress)

    boolean match;
    if UsingAArch32() && ELUsingAArch32(EL1) && PSTATE.EL != EL2 then
        // Each bit position in this string corresponds to a bit in DBGVCR and an exception vector.
        match_word = Zeros(32);

        ss = 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)

        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

        // Mask out bits not corresponding to vectors.
        bits(32) mask;
        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';

        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);

        if !IsZero(vaddress<1:0>) && match then
            match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);
    else
        match = FALSE;

    return match;

```

## Library pseudocode for aarch32/debug/authentication/AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled

```

// AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
// =====

boolean AArch32.SelfHostedSecurePrivilegedInvasiveDebugEnabled()
    // 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;
    return DBGEN == Signal\_High && SPIDEN == Signal\_High;

```

## Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

BreakpointInfo AArch32.BreakpointMatch(integer n, bits(32) vaddress, AccessDescriptor accdesc,
                                         integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert n < NumBreakpointsImplemented\(\);

    BreakpointInfo brkptinfo;
    enabled      = DBGBCR[n].E == '1';
    isbreakpnt   = TRUE;
    linked       = DBGBCR[n].BT == '0x01';
    linked_to    = FALSE;
    linked_n     = UInt(DBGBCR[n].LBN);

    state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                     linked, linked_n, isbreakpnt, accdesc);
    (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 or not this breakpoint generates a Breakpoint debug
        // event.
        (match_i, mismatch_i) = AArch32.BreakpointValueMatch(n, vaddress + 2, linked_to);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

        if value_mismatch && !mismatch_i then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    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);

        if !value_mismatch then
            value_mismatch = ConstrainUnpredictableBool(Unpredictable\_BPMISMATCHHALF);

    brkptinfo.match      = value_match && state_match && enabled;
    brkptinfo.mismatch   = value_mismatch && state_match && enabled;

    return brkptinfo;
```





```

// 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.

(boolean, boolean) AArch32.BreakpointValueMatch(integer n_in, bits(32) vaddress, boolean linked_to)

// "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.
integer n = n_in;
Constraint c;

// 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);

// 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);

dbgtype = DBGBCR[n].BT;

(c, dbgtype) = AArch32.ReservedBreakpointType(n, dbgtype);
if c == Constraint_DISABLED then return (FALSE, FALSE);
// Otherwise the dbgtype value returned by AArch32.ReservedBreakpointType is valid.

// Determine what to compare against.
match_addr      = (dbgtype == '0x0x');
mismatch        = (dbgtype == '010x');
match_vmid      = (dbgtype == '10xx');
match_cid1      = (dbgtype == 'xx1x');
match_cid2      = (dbgtype == '11xx');
linking_enabled = (dbgtype == 'xxx1');

// If called from StateMatch, is 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);

// 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);

boolean bvr_match = FALSE;
boolean bxvr_match = FALSE;

// Do the comparison.
if match_addr then
    constant integer byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned

    constant boolean byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    bvr_match = (vaddress<31:2> == DBGBCR[n]<31:2>) && byte_select_match;

elseif match_cid1 then
    bvr_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBCR[n]<31:0>);

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;

    if ELUsingAArch32(EL2) then

```

```

        vmid = ZeroExtend(VTTBR.VMID, 16);
        bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
    elseif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGXVR[n]<7:0>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGXVR[n]<15:0>;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && vmid == bvr_vmid);

elseif match_cid2 then
    bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() && !ELUsingAArch32(EL2) &&
        DBGXVR[n]<31:0> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || bxvr_match) && (!bvr_match_valid || bvr_match);

return (match && !mismatch, !match && mismatch);

```

### 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.

(Constraint, bits(4)) AArch32.ReservedBreakpointType(integer n, bits(4) bt_in)
    bits(4) bt      = bt_in;
    boolean reserved = FALSE;
    context_aware = IsContextAwareBreakpoint(n);

    // Address mismatch
    if bt == '010x' && HaltOnBreakpointOrWatchpoint() then
        reserved = TRUE;

    // Context matching
    if bt != '0x0x' && !context_aware then
        reserved = TRUE;

    // EL2 extension
    if bt == '1xxx' && !HaveEL(EL2) then
        reserved = TRUE;

    // Context matching
    if (bt IN {'011x', '11xx'}) && !IsFeatureImplemented(FEAT_VHE) &&
        !IsFeatureImplemented(FEAT_Debugv8p2) then
        reserved = TRUE;

    if reserved then
        Constraint c;
        (c, bt) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE, 4);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, bt);

```



```

// AArch32.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch32.StateMatch(bits(2) ssc_in, bit hmc_in, bits(2) pxc_in, boolean linked_in,
                           integer linked_n_in, boolean isbreakpnt, AccessDescriptor accdesc)

// "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.
bit hmc      = hmc_in;
bits(2) ssc   = ssc_in;
bits(2) pxc   = pxc_in;
boolean linked = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
// 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;
// Otherwise the hmc,ssc,pxc values are either valid or the values returned by
// CheckValidStateMatch are valid.

pl2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
pl1_match = pxc<0> == '1';
pl0_match = pxc<1> == '1';
ssu_match = isbreakpnt && hmc == '0' && pxc == '00' && ssc != '11';

boolean priv_match;
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;

// Security state match
boolean ss_match;
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

boolean linked_match = 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
        (first_ctx_cmp, last_ctx_cmp) = 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

vaddress = bits(32) UNKNOWN;
linked_to = TRUE;

```

```

(linked_match, -) = AArch32.BreakpointValueMatch(linked_n, vaddress, linked_to);

return priv_match && ss_match && (!linked || linked_match);

```

## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptions

```

// AArch32.GenerateDebugExceptions()
// =====

boolean AArch32.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch32.GenerateDebugExceptionsFrom(PSTATE.EL, ss);

```

## Library pseudocode for aarch32/debug/enables/AArch32.GenerateDebugExceptionsFrom

```

// AArch32.GenerateDebugExceptionsFrom()
// =====

boolean AArch32.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state)

    if !ELUsingAArch32(DebugTargetFrom(from_state)) then
        mask = '0'; // No PSTATE.D in AArch32 state
        return AArch64.GenerateDebugExceptionsFrom(from_el, from_state, mask);

    if DBGOSLSR.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    boolean enabled;
    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
            spd = 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();
        if from_el == ELO then enabled = enabled || SDER.SUIDEN == '1';
    else
        enabled = from_el != EL2;

    return enabled;

```

## Library pseudocode for aarch32/debug/pmu/AArch32.IncrementCycleCounter

```
// AArch32.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch32.IncrementCycleCounter()
    if !CountPMUEvents\(CYCLE\_COUNTER\_ID\) then return;
    bit d = PMCR.D; // Check divide-by-64
    bit lc = PMCR.LC;
    // Effective value of 'D' bit is 0 when Effective value of LC is '1'
    if lc == '1' then d = '0';
    if d == '1' && !HasElapsed64Cycles\(\) then return;

    constant integer old_value = UInt(PMCCNTR);
    constant integer new_value = old_value + 1;
    PMCCNTR = new_value<63:0>;

    constant integer ovflw = if lc == '1' then 64 else 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET.C = '1';
        PMOVSR.C = '1';

    return;
```

## 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.

integer AArch32.IncrementEventCounter(integer idx, integer increment_in, integer Vm)
    if HaveAArch64\(\) then
        // Force the counter to be incremented as a 64-bit counter.
        return AArch64.IncrementEventCounter(idx, increment_in, Vm);

    // 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.
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR[idx]);
    constant integer increment = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    PMEVCNTR[idx] = new_value<31:0>;
    constant integer ovflw = 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET<idx> = '1';
        PMOVSR<idx> = '1';
        // Check for the CHAIN event from an even counter
        if idx<0> == '0' && idx + 1 < NUM_PMU_COUNTERS then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);

    return increment;
```

## 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.

AArch32.PMUCycle()
    if HaveAArch64() then
        AArch64.PMUCycle();
        return;

    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    constant integer counters = NUM_PMU_COUNTERS;
    integer Vm = 0;
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                constant integer accumulated = PMUEventAccumulator[idx];
                if (idx MOD 2) == 0 then Vm = 0;
                Vm = AArch32.IncrementEventCounter(idx, accumulated, Vm);
                PMUEventAccumulator[idx] = 0;
    AArch32.IncrementCycleCounter();
    CheckForPMUOverflow();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord except)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(except);
    AArch32.WriteMode(M32_Hyp);
    SPSR_curr[] = bits(32) UNKNOWN;
    ELR_hyp = bits(32) UNKNOWN;
    // 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> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields();

    EndOfInstruction();
```



## 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.

AArch32.EnterModeInDebugState(bits(5) target_mode)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // 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> = bits(4) UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterMonitorModeInDebugState

```
// AArch32.EnterMonitorModeInDebugState()
// =====
// Take an exception in Debug state to Monitor mode.

AArch32.EnterMonitorModeInDebugState()
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = bits(32) UNKNOWN;
    R[14] = bits(32) UNKNOWN;
    // 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> = bits(4) UNKNOWN;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
    if IsFeatureImplemented(FEAT_Debugv8p9) then
        DSPSR2 = bits(32) UNKNOWN;
    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    EndOfInstruction();
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointByteMatch

```
// AArch32.WatchpointByteMatch()
// =====

boolean AArch32.WatchpointByteMatch(integer n, bits(32) vaddress)
    constant integer dbgtop = 31;
    constant integer cmpbottom = if DBGWVR[n]<2> == '1' then 2 else 3; // Word or doubleword
    bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR[n].BAS<select> != '0');
    mask = 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
        LSB = (DBGWCR[n].BAS AND NOT(DBGWCR[n].BAS - 1)); MSB = (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

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, 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

    constant integer cmpmsb = dbgtop;
    constant integer cmplsb = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (vaddress<cmpmsb:cmplsb> == DBGWVR[n]<cmpmsb:cmplsb>);
    if mask > bottom then
        // If masked bits of DBGWVR[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
        if WVR_match && !IsZero(DBGWVR[n]<cmplsb-1:bottombit>) then
            WVR_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKEDBITS);

    return (WVR_match && byte_select_match);
```

## Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

WatchpointInfo AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size,
                                         AccessDescriptor accdesc)

    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    constant boolean enabled      = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;
    linked_n = UInt(DBGWCR_EL1[n].LBN);
    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, linked_n, isbreakpnt, accdesc);

    boolean ls_match;
    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;

    boolean value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    WatchpointInfo watchptinfo;
    watchptinfo.watchpt_num = n;
    watchptinfo.value_match = value_match && state_match && ls_match && enabled;
    return watchptinfo;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(FaultRecord fault)

    // Check if routed to AArch64 state
    route_to_aarch64 = ((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'));

    if route_to_aarch64 then
        AArch64.Abort(fault);
    elsif fault.accessdesc.acctype == AccessType_IFETCH then
        AArch32.TakePrefetchAbortException(fault);
    else
        AArch32.TakeDataAbortException(fault);
```

## 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.

ExceptionRecord AArch32.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
except = ExceptionSyndrome(exceptype);

except.syndrome.iss = AArch32.FaultSyndrome(exceptype, fault);
except.vaddress = ZeroExtend(fault.vaddress, 64);

if IPAValid(fault) then
    except.ipavalid = TRUE;
    except.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
    except.ipaddress = ZeroExtend(fault.ipaddress.address, 56);
else
    except.ipavalid = FALSE;

return except;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()
    constant bits(32) pc = ThisInstrAddr(32);

    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

        constant AccessDescriptor accdesc = CreateAccDescIFetch();
        FaultRecord fault = NoFault(accdesc, ZeroExtend(pc, 64));
        // Generate an Alignment fault Prefetch Abort exception
        fault.statuscode = Fault_Alignment;
        AArch32.Abort(fault);
```

## 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.

bits(32) AArch32.CommonFaultStatus(FaultRecord fault, boolean long_format)
    bits(32) target = Zeros(32);
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        constant ErrorState errstate = PEErrState(fault);
        target<15:14> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET
    if IsExternalAbort(fault) then target<12> = fault.extflag; // 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
    return target;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportDataAbort

```
// AArch32.ReportDataAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportDataAbort(boolean route_to_monitor, FaultRecord fault)
    boolean long_format;
    if route_to_monitor && CurrentSecurityState() != SS\_Secure then
        long_format = ((TTBCR_S.EAE == '1') ||
            (IsExternalSyncAbort(fault) && ((PSTATE.EL == EL2 || TTBCR.EAE == '1') ||
                (fault.secondstage && (boolean IMPLEMENTATION_DEFINED
                    "Report abort using Long-descriptor format")))));
    else
        long_format = TTBCR.EAE == '1';
    bits(32) syndrome = 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

    if !long_format then
        syndrome<7:4> = fault.domain; // Domain

    if fault.accessdesc.acctype == AccessType\_IC then
        bits(32) i_syndrome;
        if (!long_format &&
            boolean IMPLEMENTATION_DEFINED "Report I-cache maintenance fault in IFSR") then
            i_syndrome = syndrome;
            syndrome<10,3:0> = EncodeSDFSC(Fault\_ICacheMaint, 1);
        else
            i_syndrome = bits(32) UNKNOWN;
        if route_to_monitor then
            IFSR_S = i_syndrome;
        else
            IFSR = i_syndrome;

    if route_to_monitor then
        DFSR_S = syndrome;
        DFAR_S = fault.vaddress<31:0>;
    else
        DFSR = syndrome;
        DFAR = fault.vaddress<31:0>;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.ReportPrefetchAbort

```
// AArch32.ReportPrefetchAbort()
// =====
// Report syndrome information for aborts taken to modes other than Hyp mode.

AArch32.ReportPrefetchAbort(boolean route_to_monitor, FaultRecord fault)
    // 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.
    long_format = 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';

    constant bits(32) fsr = 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>;

    return;
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(FaultRecord fault)

    bits(2) sea_target;
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = bits(2) UNKNOWN;

    route_to_monitor = IsExternalAbort(fault) && sea_target == EL3;
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (IsExternalAbort(fault) && sea_target == EL2) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    constant integer vect_offset = 0x10;
    constant integer lr_offset = 8;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    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
        except = 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);
    else
        AArch32.ReportDataAbort(route_to_monitor, fault);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(FaultRecord fault)

    bits(2) sea_target;
    if IsExternalAbort(fault) then
        sea_target = SyncExternalAbortTarget(fault);
    else
        sea_target = bits(2) UNKNOWN;

    route_to_monitor = IsExternalAbort(fault) && sea_target == EL3;
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (IsExternalAbort(fault) && sea_target == EL2) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    ExceptionRecord except;
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x0C;
    lr_offset = 4;

    if IsDebugException(fault) then DBGDSCRext.MOE = fault.debugmoe;
    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\_PCAalignment);
            except.vaddress = ThisInstrAddr(64);
        else
            except = AArch32.AbortSyndrome(Exception\_InstructionAbort, fault, EL2);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault);
        AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalFIQException

```
// AArch32.TakePhysicalFIQException()
// =====

AArch32.TakePhysicalFIQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = 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());

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.FIQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalFIQException();
    route_to_monitor = HaveEL(EL3) && SCR.FIQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.FMO == '1'));
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception\_FIQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32\_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalIRQException

```
// AArch32.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch32.TakePhysicalIRQException()

    // Check if routed to AArch64 state
    route_to_aarch64 = 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());
    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.IRQ == '1';

    if route_to_aarch64 then AArch64.TakePhysicalIRQException();

    route_to_monitor = HaveEL(EL3) && SCR.IRQ == '1';
    route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR.TGE == '1' || HCR.IMO == '1'));
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;
    if route_to_monitor then
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    elseif PSTATE.EL == EL2 || route_to_hyp then
        except = ExceptionSyndrome(Exception\_IRQ);
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    else
        AArch32.EnterMode(M32\_IRQ, preferred_exception_return, lr_offset, vect_offset);
```



## Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalErrorException

```
// AArch32.TakePhysicalErrorException()
// =====

AArch32.TakePhysicalErrorException(boolean implicit_esb)
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalErrorTarget();
    assert !masked;

    // Check if routed to AArch64 state
    if !ELUsingAArch32(target_el) then
        AArch64.TakePhysicalErrorException(implicit_esb);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    constant integer vect_offset = 0x10;
    constant integer lr_offset = 8;

    constant FaultRecord fault = GetPendingPhysicalError();
    constant bits(32) vaddress = bits(32) UNKNOWN;
    except = AArch32.AbortSyndrome(Exception\_DataAbort, fault, target_el);
    route_to_monitor = (target_el == EL3);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalError();

    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);
        when EL1
            AArch32.ReportDataAbort(route_to_monitor, fault);
            AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
        otherwise
            Unreachable();
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualFIQException

```
// AArch32.TakeVirtualFIQException()
// =====

AArch32.TakeVirtualFIQException()
    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';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x1C;
    lr_offset = 4;

    AArch32.EnterMode(M32\_FIQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualIRQException

```
// AArch32.TakeVirtualIRQException()
// =====

AArch32.TakeVirtualIRQException()
    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';

    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x18;
    lr_offset = 4;

    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualSErrorException

```
// AArch32.TakeVirtualSErrorException()
// =====

AArch32.TakeVirtualSErrorException()

    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';
    // Check if routed to AArch64 state
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualSErrorException();
    route_to_monitor = FALSE;

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    constant Fault fault = Fault_AsyncExternal;
    constant integer level = integer UNKNOWN;
    bits(32) fsr = Zeros(32);
    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;
    else
        fsr<12> = bit IMPLEMENTATION_DEFINED "Virtual External abort type";
    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);
    DFSR = fsr;
    DFAR = bits(32) UNKNOWN;
    ClearPendingVirtualSError();
    AArch32.EnterMode(M32_Abort, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/debug/AArch32.SoftwareBreakpoint

```
// AArch32.SoftwareBreakpoint()
// =====

AArch32.SoftwareBreakpoint(bits(16) immediate)

    if (EL2Enabled() && !ELUsingAArch32(EL2) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1')) || !ELUsingAArch32(EL1) then
        AArch64.SoftwareBreakpoint(immediate);

    accdesc = CreateAccDescIFetch();
    fault = NoFault(accdesc, bits(64) UNKNOWN);

    fault.statuscode = Fault\_Debug;
    fault.debugmoe = DebugException\_BKPT;

    AArch32.Abort(fault);
```

## Library pseudocode for aarch32/exceptions/debug/DebugException

```
// DebugException
// =====
// Reason codes for debug exceptions, taken to AArch32

constant bits(4) DebugException_Breakpoint = '0001';
constant bits(4) DebugException_BKPT      = '0011';
constant bits(4) DebugException_VectorCatch = '0101';
constant bits(4) DebugException_Watchpoint = '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.

AArch32.CheckAdvSIMDOrFPRegisterTraps(bits(4) reg)

    if PSTATE.EL == EL1 && EL2Enabled() then
        tid0 = if ELUsingAArch32(EL2) then HCR.TID0 else HCR_EL2.TID0;
        tid3 = if ELUsingAArch32(EL2) then HCR.TID3 else HCR_EL2.TID3;

        if ((tid0 == '1' && reg == '0000') ||
            (tid3 == '1' && reg IN {'0101', '0110', '0111'})) then // FPSID
            if ELUsingAArch32(EL2) then // MVFRx
                AArch32.SystemAccessTrap(M32\_Hyp, 0x8);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x8);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ExceptionClass

```
// AArch32.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in HSR

(integer,bit) AArch32.ExceptionClass(Exception exceptype)

    il_is_valid = TRUE;
    integer ec;
    case exceptype of
        when Exception Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception WFxTrap                ec = 0x01;
        when Exception CP15RRTTrap            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 PACTrap                ec = 0x09;
        when Exception TSTARTAccessTrap       ec = 0x1B;
        when Exception GPC                    ec = 0x1E;
        when Exception CP14RRTTrap            ec = 0x0C;
        when Exception BranchTarget           ec = 0x0D;
        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 = 0x20; il_is_valid = FALSE;
        when Exception PCAlignment            ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort              ec = 0x24;
        when Exception NV2DataAbort           ec = 0x25;
        when Exception FPTrappedException     ec = 0x28;
        when Exception Profiling              ec = 0x3D;
        otherwise                             Unreachable();

    if ec IN {0x20,0x24} && PSTATE.EL == EL2 then
        ec = ec + 1;
    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';

    return (ec,il);
```

## 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.

boolean AArch32.GeneralExceptionsToAArch64()
    return ((PSTATE.EL == EL0 && !ELUsingAArch32(EL1)) ||
            (EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1'));
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.ReportHypEntry

```
// AArch32.ReportHypEntry()
// =====
// Report syndrome information to Hyp mode registers.

AArch32.ReportHypEntry(ExceptionRecord except)

    constant Exception exceptype = except.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = except.syndrome.iss;
    iss2 = 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';

    HSR = ec<5:0>:il:iss;

    if exceptype IN {Exception_InstructionAbort, Exception_PCAalignment} then
        HIFAR = except.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = except.vaddress<31:0>;

    if except.ipavalid then
        HPFAR<31:4> = except.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

## 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.

AArch32.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch32/exceptions/exceptions/AArch32.TakeReset

```
// AArch32.TakeReset()
// =====
// Reset into AArch32 state

AArch32.TakeReset(boolean cold_reset)
    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);

    // 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.
    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();
        AArch32.ResetSpecialRegisters();
        ResetExternalDebugRegisters(cold_reset);

    bits(32) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL(EL3) then
        if MVBAR<0> == '1' then // Reset vector in MVBAR
            rv = MVBAR<31:1>:'0';
        else
            rv = bits(32) IMPLEMENTATION_DEFINED "reset vector address";
    else
        rv = RVBAR<31:1>:'0';

    // The reset vector must be correctly aligned
    assert rv<0> == '0' && (PSTATE.T == '1' || rv<1> == '0');

    constant boolean branch_conditional = FALSE;
    EDPRSR.R = '0'; // Leaving Reset State.
    BranchTo(rv, BranchType_RESET, branch_conditional);
```

## Library pseudocode for aarch32/exceptions/exceptions/ExcVectorBase

```
// ExcVectorBase()
// =====

bits(32) ExcVectorBase()
    if SCTLR.V == '1' then // Hivecs selected, base = 0xFFFF0000
        return Ones(16):Zeros(16);
    else
        return VBAR<31:5>:Zeros(5);
```

## Library pseudocode for aarch32/exceptions/ieeefp/AArch32.FPTrappedException

```
// AArch32.FPTrappedException()
// =====

AArch32.FPTrappedException(bits(8) accumulated_exceptions)
    if AArch32.GeneralExceptionsToAArch64() then
        is_ase = FALSE;
        element = 0;
        AArch64.FPTrappedException(is_ase, accumulated_exceptions);
    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();
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallHypervisor

```
// AArch32.CallHypervisor()
// =====
// Performs a HVC call

AArch32.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if !ELUsingAArch32(EL2) then
        AArch64.CallHypervisor(immediate);
    else
        AArch32.TakeHVCEXception(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.CallSupervisor

```
// AArch32.CallSupervisor()
// =====
// Calls the Supervisor

AArch32.CallSupervisor(bits(16) immediate_in)
    bits(16) immediate = immediate_in;
    if AArch32.CurrentCond() != '1110' then
        immediate = bits(16) UNKNOWN;
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CallSupervisor(immediate);
    else
        AArch32.TakeSVCEXception(immediate);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeHVCEException

```
// AArch32.TakeHVCEException()
// =====

AArch32.TakeHVCEException(bits(16) immediate)
    assert HaveEL(EL2) && ELUsingAArch32(EL2);

    AArch32.ITAdvance();
    SSAdvance();
    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;

    except = 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);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCEException

```
// AArch32.TakeSMCEException()
// =====

AArch32.TakeSMCEException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    HSAdvance();
    SSAdvance();
    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCEException

```
// AArch32.TakeSVCEException()
// =====

AArch32.TakeSVCEException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    constant bits(32) preferred_exception_return = NextInstrAddr(32);
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        except = 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);
    else
        AArch32.EnterMode(M32_Svc, preferred_exception_return, lr_offset, vect_offset);
```



## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterHypMode

```
// AArch32.EnterHypMode()
// =====
// Take an exception to Hyp mode.

AArch32.EnterHypMode(ExceptionRecord except, bits(32) preferred_exception_return,
                     integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    if Halted() then
        AArch32.EnterHypModeInDebugState(except);
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32.NonDebugState, 32);
    if ! except.excepttype IN {Exception_IRQ, Exception_FIQ} then
        AArch32.ReportHypEntry(except);
    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) || SCR_curr[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_curr[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_curr[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = HSCTLR.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(HVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMode

```
// AArch32.EnterMode()
// =====
// Take an exception to a mode other than Monitor and Hyp mode.

AArch32.EnterMode(bits(5) target_mode, bits(32) preferred_exception_return, integer lr_offset,
                  integer vect_offset)
    SynchronizeContext();
    assert ELUsingAArch32(EL1) && PSTATE.EL != EL2;

    if Halted() then
        AArch32.EnterModeInDebugState(target_mode);
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32.NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.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';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTL.R.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(ExcVectorBase()<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/takeexception/AArch32.EnterMonitorMode

```
// AArch32.EnterMonitorMode()
// =====
// Take an exception to Monitor mode.

AArch32.EnterMonitorMode(bits(32) preferred_exception_return, integer lr_offset,
                          integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    from_secure = CurrentSecurityState() == SS_Secure;
    if Halted() then
        AArch32.EnterMonitorModeInDebugState();
        return;
    constant bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState, 32);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR_curr[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTL.R.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if IsFeatureImplemented(FEAT_PAN) then
        if !from_secure then
            PSTATE.PAN = '0';
        elseif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTL.R.DSSBS;
    constant boolean branch_conditional = FALSE;
    BranchTo(MVBAR<31:5>:vect_offset<4:0>, BranchType_EXCEPTION, branch_conditional);

    CheckExceptionCatch(TRUE); // Check for debug event on exception entry

    EndOfInstruction();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckAdvSIMDOrFPEEnabled

```
// AArch32.CheckAdvSIMDOrFPEEnabled()
// =====
// Check against CPACR, FPEXC, HCPTR, NSACR, and CPTR_EL3.

AArch32.CheckAdvSIMDOrFPEEnabled(boolean fpexc_check_in, boolean advsimd)
    boolean fpexc_check = 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
        cpacr_asedis = CPACR.ASEDIS;
        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';
            if NSACR.cp10 == '0' then cpacr_cp10 = '00';

        if PSTATE.EL != EL2 then
            // Check if Advanced SIMD disabled in CPACR
            if advsimd && cpacr_asedis == '1' then AArch32.Undefined\(\);

            // Check if access disabled in CPACR
            boolean disabled;
            case cpacr_cp10 of
                when '00' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0;
                when '10' disabled = ConstrainUnpredictableBool\(Unpredictable\_RESCPACR\);
                when '11' disabled = FALSE;
            if disabled then AArch32.Undefined\(\);

        // 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 = (boolean IMPLEMENTATION_DEFINED
                "Use FPEXC32_EL2.EN value when {TGE,RW} == {1,0}");

        if fpexc_check && FPEXC.EN == '0' then
            AArch32.Undefined\(\);

        AArch32.CheckFPAdvSIMDTrap(advsimd);    // Also check against HCPTR and CPTR_EL3
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckFPAdvSIMDTrap

```
// AArch32.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch32.CheckFPAdvSIMDTrap(boolean advsimd)
    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64.CheckFPAdvSIMDTrap\(\);
    else
        if (HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) &&
            CPTR_EL3.TFP == '1' && EL3SDDUndefPriority\(\)) then
            UNDEFINED;

        ss = CurrentSecurityState\(\);
        if HaveEL\(EL2\) && ss != SS\_Secure then
            hcptr_tase = HCPTR.TASE;
            hcptr_cp10 = HCPTR.TCP10;

            if HaveEL\(EL3\) && ELUsingAArch32\(EL3\) then
                // Check if access disabled in NSACR
                if NSACR.NSASEDIS == '1' then hcptr_tase = '1';
                if NSACR.cp10 == '0' then hcptr_cp10 = '1';

            // Check if access disabled in HCPTR
            if (advsimd && hcptr_tase == '1') || hcptr_cp10 == '1' then
                except = 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

                if PSTATE.EL == EL2 then
                    AArch32.TakeUndefInstrException(except);
                else
                    AArch32.TakeHypTrapException(except);

            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\(EL3\);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSMCUndefOrTrap

```
// AArch32.CheckForSMCUndefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch32.CheckForSMCUndefOrTrap()
    if !HaveEL\(EL3\) || PSTATE.EL == EL0 then
        UNDEFINED;

    if EL2Enabled\(\) && !ELUsingAArch32\(EL2\) then
        AArch64.CheckForSMCUndefOrTrap\(Zeros\(16\)\);
    else
        route_to_hyp = EL2Enabled\(\) && PSTATE.EL == EL1 && HCR.TSC == '1';
        if route_to_hyp then
            except = ExceptionSyndrome\(Exception\_MonitorCall\);
            AArch32.TakeHypTrapException(except);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                           (!IsInHost() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception\_SupervisorCall);
            except.syndrome.iss<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            constant bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForWfxTrap

```
// AArch32.CheckForWfxTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch32.CheckForWfxTrap(bits(2) target_el, WfxType wfxtype)
    assert HaveEL(target_el);

    // Check for routing to AArch64
    if !ELUsingAArch32(target_el) then
        boolean trap;
        constant boolean is_wfe = wfxtype == WfxType_WFE;
        case target_el of
            when EL1
                trap = (if is_wfe then SCTL_ELx[].nTWE else SCTL_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';

        if trap then
            if target_el == EL3 && EL3SDDUndef() then
                UNDEFINED;
            else
                AArch64.WfxTrap(wfxtype, target_el);
        return;

    constant boolean is_wfe = wfxtype == WfxType_WFE;
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTL.nTWE else SCTL.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';

    if trap then
        if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
            AArch64.WfxTrap(wfxtype, target_el);

        if target_el == EL3 && !EL3SDDUndef() then
            AArch32.TakeMonitorTrapException();
        elsif target_el == EL2 then
            except = 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();
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckITEnabled

```
// AArch32.CheckITEnabled()
// =====
// Check whether the T32 IT instruction is disabled.

AArch32.CheckITEnabled(bits(4) mask)
    bit it_disabled;
    if PSTATE.EL == EL2 then
        it_disabled = HSCTLR.ITD;
    else
        it_disabled = (if ELUsingAArch32(EL1) then SCTLR.ITD else SCTLR_ELx[].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    accdesc = CreateAccDescIFetch();
    aligned = TRUE;
    // Otherwise whether the IT block is allowed depends on hwl of the next instruction.
    next_instr = AArch32.MemSingle(NextInstrAddr(32), 2, accdesc, aligned);

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                     '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxx111'} 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;

return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckIllegalState

```
// AArch32.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch32.CheckIllegalState()
    if AArch32.GeneralExceptionsToAArch64() then
        AArch64.CheckIllegalState();
    elseif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        constant bits(32) preferred_exception_return = ThisInstrAddr(32);
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            except = ExceptionSyndrome(Exception\_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
        else
            AArch32.TakeUndefInstrException();
```



## Library pseudocode for aarch32/exceptions/traps/AArch32.CheckSETENDEnabled

```
// AArch32.CheckSETENDEnabled()
// =====
// Check whether the AArch32 SETEND instruction is disabled.

AArch32.CheckSETENDEnabled()
    bit setend_disabled;
    if PSTATE.EL == EL2 then
        setend_disabled = HSCTLR.SED;
    else
        setend_disabled = (if ELUsingAArch32(EL1) then SCTLR.SED else SCTLR_ELx[].SED);
    if setend_disabled == '1' then
        UNDEFINED;

    return;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.SystemAccessTrap

```
// AArch32.SystemAccessTrap()
// =====
// Trapped System register access.

AArch32.SystemAccessTrap(bits(5) mode, integer ec)
    (valid, target_el) = ELFromM32(mode);
    assert valid && HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    if target_el == EL2 then
        except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(except);
    else
        AArch32.TakeUndefInstrException();
```

## 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.

ExceptionRecord AArch32.SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    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();

    bits(20) iss = Zeros(20);

    if except.excepttype == Exception Uncategorized then
        return except;
    elseif except.excepttype 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.excepttype != 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
    elseif except.excepttype 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.excepttype == 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> = bits(4) UNKNOWN;
            iss<3>   = '1';
        iss<0> = instr<20>;                 // Direction

    except.syndrome.iss<24:20> = ConditionSyndrome();
    except.syndrome.iss<19:0>  = iss;

    return except;
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    except = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch32.TakeHypTrapException(except);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(ExceptionRecord except)
    assert HaveEL(EL2) && CurrentSecurityState() == SS\_NonSecure && ELUsingAArch32(EL2);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x14;

    AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeMonitorTrapException

```
// AArch32.TakeMonitorTrapException()
// =====
// Exceptions routed to Monitor mode as a Monitor Trap exception.

AArch32.TakeMonitorTrapException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);

    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.TakeUndefInstrException

```
// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException()
    except = ExceptionSyndrome(Exception Uncategorized);
    AArch32.TakeUndefInstrException(except);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord except)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';
    constant bits(32) preferred_exception_return = ThisInstrAddr(32);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet() == InstrSet\_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(except, preferred_exception_return, vect_offset);
    elsif route_to_hyp then
        AArch32.EnterHypMode(except, preferred_exception_return, 0x14);
    else
        AArch32.EnterMode(M32\_Undef, preferred_exception_return, lr_offset, vect_offset);
```

## Library pseudocode for aarch32/exceptions/traps/AArch32.Undefined

```
// AArch32.Undefined()
// =====

AArch32.Undefined()

    if AArch32.GeneralExceptionsToAArch64\(\) then AArch64.Undefined\(\);
    AArch32.TakeUndefInstrException\(\);
```

## Library pseudocode for aarch32/functions/aborts/AArch32.DomainValid

```
// AArch32.DomainValid()
// =====
// Returns TRUE if the Domain is valid for a Short-descriptor translation scheme.

boolean AArch32.DomainValid(Fault statuscode, integer level)
    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;
```

## 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.

bits(25) AArch32.FaultSyndrome(Exception exceptype, FaultRecord fault)
    assert fault.statuscode != Fault_None;

    IssType isstype;
    isstype.iss = Zeros(25);

    constant boolean d_side = exceptype == Exception_DataAbort;
    if IsFeatureImplemented(FEAT_RAS) && IsAsyncAbort(fault) then
        constant ErrorState errstate = PEErrState(fault);
        isstype.iss<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fslwalk &&
            (!IsExternalSyncAbort(fault) ||
            (!IsFeatureImplemented(FEAT_RAS) && fault.accessdesc.acctype == AccessType_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
            isstype.iss<24:14> = LSInstructionSyndrome();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            isstype.iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss<6> = bit UNKNOWN;
        else
            isstype.iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then isstype.iss<9> = fault.extflag;
    isstype.iss<7> = if fault.s2fslwalk then '1' else '0';
    isstype.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return isstype.iss;
```

## Library pseudocode for aarch32/functions/aborts/EncodeSDFSC

```
// EncodeSDFSC()
// =====
// Function that gives the Short-descriptor FSR code for different types of Fault

bits(5) EncodeSDFSC(Fault statuscode, integer level)

    bits(5) result;
    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();

    return result;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm

```
// A32ExpandImm()
// =====

bits(32) A32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = A32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```

## Library pseudocode for aarch32/functions/common/A32ExpandImm\_C

```
// A32ExpandImm_C()
// =====

(bits(32), bit) A32ExpandImm_C(bits(12) imm12, bit carry_in)

    unrotated_value = ZeroExtend(imm12<7:0>, 32);
    (imm32, carry_out) = Shift_C(unrotated_value, SRTYPE_ROR, 2*UInt(imm12<11:8>), carry_in);

    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/common/DecodeImmShift

```
// DecodeImmShift()
// =====

(SRTYPE, integer) DecodeImmShift(bits(2) srtype, bits(5) imm5)

    SRTYPE shift_t;
    integer shift_n;
    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);

    return (shift_t, shift_n);
```

## Library pseudocode for aarch32/functions/common/DecodeRegShift

```
// DecodeRegShift()
// =====

SRTYPE DecodeRegShift(bits(2) srtype)

    SRTYPE shift_t;
    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;

    return shift_t;
```

## Library pseudocode for aarch32/functions/common/RRX

```
// RRX()
// =====

bits(N) RRX(bits(N) x, bit carry_in)

    (result, -) = RRX_C(x, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/RRX\_C

```
// RRX_C()
// =====

(bits(N), bit) RRX_C(bits(N) x, bit carry_in)
    result = carry_in : x<N-1:1>;
    carry_out = x<0>;
    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/SRType

```
// SRType
// =====

enumeration SRType {SRType_LSL, SRType_LSR, SRType_ASR, SRType_ROR, SRType_RRX};
```

## Library pseudocode for aarch32/functions/common/Shift

```
// Shift()
// =====

bits(N) Shift(bits(N) value, SRType srtype, integer amount, bit carry_in)
    (result, -) = Shift\_C(value, srtype, amount, carry_in);
    return result;
```

## Library pseudocode for aarch32/functions/common/Shift\_C

```
// Shift_C()
// =====

(bits(N), bit) Shift_C(bits(N) value, SRType srtype, integer amount, bit carry_in)
    assert !(srtype == SRType\_RRX && amount != 1);

    bits(N) result;
    bit carry_out;
    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(value, carry_in);

    return (result, carry_out);
```

## Library pseudocode for aarch32/functions/common/T32ExpandImm

```
// T32ExpandImm()
// =====

bits(32) T32ExpandImm(bits(12) imm12)

    // PSTATE.C argument to following function call does not affect the imm32 result.
    (imm32, -) = T32ExpandImm\_C(imm12, PSTATE.C);

    return imm32;
```



## Library pseudocode for aarch32/functions/common/T32ExpandImm\_C

```
// T32ExpandImm_C()
// =====

(bits(32), bit) T32ExpandImm_C(bits(12) imm12, bit carry_in)
    bits(32) imm32;
    bit carry_out;
    if imm12<11:10> == '00' then
        case imm12<9:8> of
            when '00'
                imm32 = ZeroExtend(imm12<7:0>, 32);
            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>;
        carry_out = carry_in;
    else
        unrotated_value = ZeroExtend('1':imm12<6:0>, 32);
        (imm32, carry_out) = ROR_C(unrotated_value, UInt(imm12<11:7>));

    return (imm32, carry_out);
```

## Library pseudocode for aarch32/functions/common/VBitOps

```
// VBitOps
// =====

enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

## Library pseudocode for aarch32/functions/common/VCGEType

```
// VCGEType
// =====

enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

## Library pseudocode for aarch32/functions/common/VCGTtype

```
// VCGTtype
// =====

enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

## Library pseudocode for aarch32/functions/common/VFPNegMul

```
// VFPNegMul
// =====

enumeration VFPNegMul {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.

AArch32.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    if PSTATE.EL == EL0 && (!ELUsingAArch32\(EL1\) ||
        (EL2Enabled\(\) && !ELUsingAArch32\(EL2\))) then
        AArch64.CheckCP15InstrCoarseTraps(CRn, nreg, CRm);

    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
        major = if nreg == 1 then CRn else CRm;
        // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR<CRn/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 &&
                boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
                UNDEFINED;
            if ELUsingAArch32\(EL2\) then
                AArch32.SystemAccessTrap(M32_Hyp, 0x3);
            else
                AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```

## 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 an memory abort.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp_STORE, acqrel,
                                                            tagchecked, privileged);

    constant boolean aligned = IsAligned(address, size);

    if !aligned then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    if !AArch32.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

## 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.

boolean AArch32.IsExclusiveVA(bits(32) address, integer processorid, integer size);
```

## 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.

AArch32.MarkExclusiveVA(bits(32) address, integer processorid, integer size);
```

## 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.

AArch32.SetExclusiveMonitors(bits(32) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel,
                                                    tagchecked, privileged);

    constant boolean aligned = IsAligned(address, size);

    if !aligned then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch32.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDEnabled

```
// CheckAdvSIMDEnabled()
// =====

CheckAdvSIMDEnabled()

    fpexc_check = TRUE;
    advsimd = TRUE;

    AArch32.CheckAdvSIMDOrFPEEnabled(fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() 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
        _Dclone[i] = D[i];

    return;
```

## Library pseudocode for aarch32/functions/float/CheckAdvSIMDOrVFPEEnabled

```
// CheckAdvSIMDOrVFPEEnabled()
// =====

CheckAdvSIMDOrVFPEEnabled(boolean include_fpexc_check, boolean advsimd)
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckCryptoEnabled32

```
// CheckCryptoEnabled32()
// =====

CheckCryptoEnabled32()
    CheckAdvSIMDEnabled();
    // Return from CheckAdvSIMDEnabled() occurs only if access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/CheckVFPEEnabled

```
// CheckVFPEEnabled()
// =====

CheckVFPEEnabled(boolean include_fpexc_check)
    advsimd = FALSE;
    AArch32.CheckAdvSIMDOrFPEEnabled(include_fpexc_check, advsimd);
    // Return from CheckAdvSIMDOrFPEEnabled() occurs only if VFP access is permitted
    return;
```

## Library pseudocode for aarch32/functions/float/FPHalvedSub

```
// FPHalvedSub()
// =====

bits(N) FPHalvedSub(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity);  inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);      zero2 = (type2 == FPType\_Zero);
        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 != sign2 then
            result = FPZero(sign1, N);
        else
            result_value = (value1 - value2) / 2.0;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, N);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    constant FPCR\_Type fpcr = StandardFPCR();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity);  inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);      zero2 = (type2 == FPTYPE\_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        constant bits(N) three = FPThree('0', N);
        result = FPHalvedSub(three, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/FPRecipStep

```
// FPRecipStep()
// =====

bits(N) FPRecipStep(bits(N) op1, bits(N) op2)
    assert N IN {16,32};
    constant FPCR\_Type fpcr = StandardFPCR();
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity);  inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);      zero2 = (type2 == FPTYPE\_Zero);
        bits(N) product;
        if (inf1 && zero2) || (zero1 && inf2) then
            product = FPZero('0', N);
        else
            product = FPMul(op1, op2, fpcr);
        constant bits(N) two = FPTwo('0', N);
        result = FPSub(two, product, fpcr);
    return result;
```

## Library pseudocode for aarch32/functions/float/StandardFPCR

```
// StandardFPCR()
// =====

FPCR_Type StandardFPCR()
    constant bits(32) value = '00000' : FPCR.AHP : '110000' : FPCR.FZ16 : '00000000000000000000';
    return ZeroExtend(value, 64);
```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingle

```
// AArch32.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc,
                                boolean aligned]
    bits(size*8) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (value, memaddrdesc, memstatus) = AArch32.MemSingleRead(address, size, accdesc, aligned);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, size, accdesc);

    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (memaddrdesc, memstatus) = AArch32.MemSingleWrite(address, size, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch32.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingleRead

```
// AArch32.MemSingleRead()
// =====
// Perform an atomic, little-endian read of 'size' bytes.

(bits(size*8), AddressDescriptor, PhysMemRetStatus) AArch32.MemSingleRead(bits(32) address,
                                                                    integer size,
                                                                    AccessDescriptor accdesc_in,
                                                                    boolean aligned)

assert size IN {1, 2, 4, 8, 16, 32};
bits(size*8) value = bits(size*8) UNKNOWN;
PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
constant AccessDescriptor accdesc = accdesc_in;
assert IsAligned(address, size);

AddressDescriptor memaddrdesc;
memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (value, memaddrdesc, memstatus);

// Memory array access
(memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
if IsFault(memstatus) then
    return (value, memaddrdesc, memstatus);

return (value, memaddrdesc, memstatus);
```

## Library pseudocode for aarch32/functions/memory/AArch32.MemSingleWrite

```
// AArch32.MemSingleWrite()
// =====
// Perform an atomic, little-endian write of 'size' bytes.

(AddressDescriptor, PhysMemRetStatus) AArch32.MemSingleWrite(bits(32) address, integer size,
                                                                    AccessDescriptor accdesc_in,
                                                                    boolean aligned, bits(size*8) value)

assert size IN {1, 2, 4, 8, 16, 32};
constant AccessDescriptor accdesc = accdesc_in;
assert IsAligned(address, size);

AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
memaddrdesc = AArch32.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (memaddrdesc, memstatus);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
if IsFault(memstatus) then
    return (memaddrdesc, memstatus);

return (memaddrdesc, memstatus);
```



## Library pseudocode for aarch32/functions/memory/AArch32.UnalignedAccessFaults

```
// AArch32.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch32.UnalignedAccessFaults(AccessDescriptor accdesc)
    return (AlignmentEnforced() ||
           accdesc.a32lsmd ||
           accdesc.exclusive ||
           accdesc.acqsc ||
           accdesc.relsc);
```

## Library pseudocode for aarch32/functions/memory/Hint\_PreloadData

```
// Hint_PreloadData()
// =====

Hint_PreloadData(bits(32) address);
```

## Library pseudocode for aarch32/functions/memory/Hint\_PreloadDataForWrite

```
// Hint_PreloadDataForWrite()
// =====

Hint_PreloadDataForWrite(bits(32) address);
```

## Library pseudocode for aarch32/functions/memory/Hint\_PreloadInstr

```
// Hint_PreloadInstr()
// =====

Hint_PreloadInstr(bits(32) address);
```

## Library pseudocode for aarch32/functions/memory/MemA

```
// MemA[] - non-assignment form
// =====

bits(8*size) MemA(bits(32) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel, tagchecked,
                                                         privileged);

    return Mem\_with\_type[address, size, accdesc];

// MemA[] - assignment form
// =====

MemA(bits(32) address, integer size) = bits(8*size) value
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_STORE, acqrel, tagchecked,
                                                         privileged);

    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO(bits(32) address, integer size)
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp\_LOAD, tagchecked);
    return Mem\_with\_type[address, size, accdesc];

// MemO[] - assignment form
// =====

MemO(bits(32) address, integer size) = bits(8*size) value
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescAcqRel(MemOp\_STORE, tagchecked);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemS

```
// MemS[] - non-assignment form
// =====
// Memory accessor for streaming load multiple instructions

bits(8*size) MemS(bits(32) address, integer size)
    constant AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp\_LOAD);
    return Mem\_with\_type[address, size, accdesc];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS(bits(32) address, integer size) = bits(8*size) value
    constant AccessDescriptor accdesc = CreateAccDescA32LSMD(MemOp\_STORE);
    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU(bits(32) address, integer size)
    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged,
                                                         tagchecked);

    return Mem\_with\_type[address, size, accdesc];

// MemU[] - assignment form
// =====

MemU(bits(32) address, integer size) = bits(8*size) value
    constant boolean nontemporal = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged,
                                                         tagchecked);

    Mem\_with\_type[address, size, accdesc] = value;
    return;
```

## Library pseudocode for aarch32/functions/memory/MemU\_unpriv

```
// MemU_unpriv[] - non-assignment form
// =====

bits(8*size) MemU_unpriv[bits(32) address, integer size]
    constant boolean nontemporal = FALSE;
    constant boolean privileged = FALSE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_LOAD, nontemporal, privileged,
                                                    tagchecked);

    return Mem\_with\_type[address, size, accdesc];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    constant boolean nontemporal = FALSE;
    constant boolean privileged = FALSE;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescGPR(MemOp\_STORE, nontemporal, privileged,
                                                    tagchecked);

    Mem\_with\_type[address, size, accdesc] = value;
    return;
```



```

// Mem_with_type[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch32.MemSingle directly.

bits(size*8) Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc_in]
    assert size IN {1, 2, 4, 8, 16, 32};
    constant AccessDescriptor accdesc = accdesc_in;
    bits(size * 8) value;
    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = if accdesc.ispair then size DIV 2 else size;
    boolean aligned    = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);
    if aligned then
        value = AArch32.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, 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.
        c = ConstrainUnpredictable(Unpredictable DEVPAGE2);
        assert c IN {Constraint FAULT, Constraint NONE};
        if c == Constraint NONE then aligned = TRUE;

        for i = 1 to size-1
            Elem[value, i, 8] = AArch32.MemSingle[address+i, 1, accdesc, aligned];

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return value;

// Mem_with_type[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.

Mem_with_type[bits(32) address, integer size, AccessDescriptor accdesc_in] = bits(size*8) value_in
    bits(size*8) value = value_in;
    constant AccessDescriptor accdesc = accdesc_in;

    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = if accdesc.ispair then size DIV 2 else size;
    boolean aligned    = IsAligned(address, alignment);

    if !aligned && AArch32.UnalignedAccessFaults(accdesc) then
        constant FaultRecord fault = AlignmentFault(accdesc, ZeroExtend(address, 64));
        AArch32.Abort(fault);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);
    if aligned then
        AArch32.MemSingle[address, size, accdesc, aligned] = value;
    else
        assert size > 1;
        AArch32.MemSingle[address, 1, 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.

        c = ConstrainUnpredictable(Unpredictable DEVPAGE2);
        assert c IN {Constraint FAULT, Constraint NONE};
        if c == Constraint NONE then aligned = TRUE;

        for i = 1 to size-1

```

```

    AArch32.MemSingle[address+i, 1, accdesc, aligned] = Elem[value, i, 8];
return;

```

## Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state.

AArch32.ESBOperation()
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();

    // Check if routed to AArch64 state
    if !masked && !ELUsingAArch32(target_el) then
        AArch64.ESBOperation();
        return;

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1'; // A
        syndrome<15:0> = AArch32.PhysicalErrorSyndrome();
        DISR = syndrome;
        ClearPendingPhysicalSError();

    return;

```

## Library pseudocode for aarch32/functions/ras/AArch32.EncodeAsyncErrorSyndrome

```

// AArch32.EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch32 state.

bits(2) AArch32.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    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();

```

## Library pseudocode for aarch32/functions/ras/AArch32.PhysicalErrorSyndrome

```

// AArch32.PhysicalErrorSyndrome()
// =====
// Generate SError syndrome.

bits(16) AArch32.PhysicalErrorSyndrome()
    bits(32) syndrome = Zeros(32);
    constant FaultRecord fault = GetPendingPhysicalSError();
    if PSTATE.EL == EL2 then
        constant ErrorState errstate = PEErrorState(fault);
        syndrome<11:10> = AArch32.EncodeAsyncErrorSyndrome(errstate); // AET
        syndrome<9> = fault.extflag; // EA
        syndrome<5:0> = '010001'; // DFSC
    else
        constant boolean long_format = TTBCR.EAE == '1';
        syndrome = AArch32.CommonFaultStatus(fault, long_format);
    return syndrome<15:0>;

```

## 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.

AArch32.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESBOperation();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a virtual
    // SError interrupt might be pending.
    vsei_pending = IsVirtualSErrorPending() && HCR.TGE == '0' && HCR.AMO == '1';
    vsei_masked = PSTATE.A == '1' || Halted() || ExternalDebugInterruptsDisabled(EL1);

    // Check for a masked virtual SError pending
    if vsei_pending && vsei_masked then
        bits(32) syndrome = Zeros(32);
        syndrome<31> = '1'; // A
        syndrome<15:14> = VDFSR<15:14>; // AET
        syndrome<12> = VDFSR<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
        VDISR = syndrome;
        ClearPendingVirtualSError();
    elseif IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) then
        AArch64.dESBOperation();

    return;
```

## Library pseudocode for aarch32/functions/registers/AArch32.ResetGeneralRegisters

```
// AArch32.ResetGeneralRegisters()
// =====

AArch32.ResetGeneralRegisters()

    for i = 0 to 7
        R[i] = bits(32) UNKNOWN;
    for i = 8 to 12
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
    if HaveEL(EL2) then Rmode[13, M32_Hyp] = bits(32) UNKNOWN; // No R14_hyp
    for i = 13 to 14
        Rmode[i, M32_User] = bits(32) UNKNOWN;
        Rmode[i, M32_FIQ] = bits(32) UNKNOWN;
        Rmode[i, M32_IRQ] = bits(32) UNKNOWN;
        Rmode[i, M32_Svc] = bits(32) UNKNOWN;
        Rmode[i, M32_Abort] = bits(32) UNKNOWN;
        Rmode[i, M32_Undef] = bits(32) UNKNOWN;
        if HaveEL(EL3) then Rmode[i, M32_Monitor] = bits(32) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSIMDFPRegisters

```
// AArch32.ResetSIMDFPRegisters()
// =====

AArch32.ResetSIMDFPRegisters()

    for i = 0 to 15
        Q[i] = bits(128) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSpecialRegisters

```
// AArch32.ResetSpecialRegisters()
// =====

AArch32.ResetSpecialRegisters()

    // AArch32 special registers
    SPSR_fiq<31:0> = bits(32) UNKNOWN;
    SPSR_irq<31:0> = bits(32) UNKNOWN;
    SPSR_svc<31:0> = bits(32) UNKNOWN;
    SPSR_abt<31:0> = bits(32) UNKNOWN;
    SPSR_und<31:0> = bits(32) UNKNOWN;
    if HaveEL(EL2) then
        SPSR_hyp = bits(32) UNKNOWN;
        ELR_hyp = bits(32) UNKNOWN;
    if HaveEL(EL3) then
        SPSR_mon = bits(32) UNKNOWN;

    // External debug special registers
    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;

    return;
```

### Library pseudocode for aarch32/functions/registers/AArch32.ResetSystemRegisters

```
// AArch32.ResetSystemRegisters()
// =====

AArch32.ResetSystemRegisters(boolean cold_reset);
```

### Library pseudocode for aarch32/functions/registers/ALUEExceptionReturn

```
// ALUEExceptionReturn()
// =====

ALUEExceptionReturn(bits(32) address)
    if PSTATE.EL == EL2 then
        UNDEFINED;
    elsif PSTATE.M IN {M32 User, M32 System} then
        constant Constraint c = ConstrainUnpredictable(Unpredictable\_ALUEXCEPTIONRETURN);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP};
        case c of
            when Constraint\_UNDEF
                UNDEFINED;
            when Constraint\_NOP
                ExecuteAsNOP();
        else
            AArch32.ExceptionReturn(address, SPSR\_curr[]);
```



## Library pseudocode for aarch32/functions/registers/ALUWritePC

```
// ALUWritePC()
// =====

ALUWritePC(bits(32) address)
  if CurrentInstrSet\(\) == InstrSet\_A32 then
    BXWritePC(address, BranchType\_INDIR);
  else
    BranchWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/BXWritePC

```
// BXWritePC()
// =====

BXWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = 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 processor 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';
  constant boolean branch_conditional = AArch32.CurrentCond() != '111x';
  BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/BranchWritePC

```
// BranchWritePC()
// =====

BranchWritePC(bits(32) address_in, BranchType branch_type)
  bits(32) address = address_in;
  if CurrentInstrSet() == InstrSet\_A32 then
    address<1:0> = '00';
  else
    address<0> = '0';
  constant boolean branch_conditional = AArch32.CurrentCond() != '111x';
  BranchTo(address, branch_type, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/CBWritePC

```
// CBWritePC()
// =====
// Takes a branch from a CBNZ/CBZ instruction.

CBWritePC(bits(32) address_in)
  bits(32) address = address_in;
  assert CurrentInstrSet() == InstrSet\_T32;
  address<0> = '0';
  constant boolean branch_conditional = TRUE;
  BranchTo(address, BranchType\_DIR, branch_conditional);
```

## Library pseudocode for aarch32/functions/registers/D

```
// D[] - non-assignment form
// =====

bits(64) D[integer n]
    assert n >= 0 && n <= 31;
    constant bits(128) vreg = V[n DIV 2, 128];
    return Elem[vreg, n MOD 2, 64];

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 2, 128];
    Elem[vreg, n MOD 2, 64] = value;
    V[n DIV 2, 128] = vreg;
    return;
```

## Library pseudocode for aarch32/functions/registers/Din

```
// Din[] - non-assignment form
// =====

bits(64) Din[integer n]
    assert n >= 0 && n <= 31;
    return \_Dclone[n];
```

## Library pseudocode for aarch32/functions/registers/H

```
// H[] - non-assignment form
// =====

bits(16) H[integer n]
    assert n >= 0 && n <= 31;
    return S[n]<15:0>;

// H[] - assignment form
// =====

H[integer n] = bits(16) value
    S[n] = ZeroExtend(value, 32);
```

## Library pseudocode for aarch32/functions/registers/LR

```
// LR - assignment form
// =====

LR = bits(32) value
    R[14] = value;
    return;

// LR - non-assignment form
// =====

bits(32) LR
    return R[14];
```

## Library pseudocode for aarch32/functions/registers/LoadWritePC

```
// LoadWritePC()
// =====

LoadWritePC(bits(32) address)
    BXWritePC(address, BranchType\_INDIR);
```

## Library pseudocode for aarch32/functions/registers/LookUpRIndex

```
// LookUpRIndex()
// =====

integer LookUpRIndex(integer n, bits(5) mode)
    assert n >= 0 && n <= 14;

    integer result;
    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;

    return result;
```

## 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

bits(32) SP_mon;
bits(32) LR_mon;
```

## Library pseudocode for aarch32/functions/registers/PC32

```
// AArch32 program counter

// PC32 - non-assignment form
// =====

bits(32) PC32
    return R[15]; // This includes the offset from AArch32 state
```

## Library pseudocode for aarch32/functions/registers/PCStoreValue

```
// PCStoreValue()
// =====

bits(32) PCStoreValue()
    // 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;
```

## Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n, 128];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n, 128] = value;
    return;
```

## Library pseudocode for aarch32/functions/registers/Qin

```
// Qin[] - non-assignment form
// =====

bits(128) Qin[integer n]
    assert n >= 0 && n <= 15;
    return Din[2*n+1]:Din[2*n];
```

## Library pseudocode for aarch32/functions/registers/R

```
// R[] - assignment form
// =====

R[integer n] = bits(32) value
    Rmode[n, PSTATE.M] = value;
    return;

// R[] - non-assignment form
// =====

bits(32) R[integer n]
    if n == 15 then
        offset = (if CurrentInstrSet() == InstrSet\_A32 then 8 else 4);
        return _PC<31:0> + offset;
    else
        return Rmode[n, PSTATE.M];
```

## Library pseudocode for aarch32/functions/registers/RBankSelect

```
// RBankSelect()
// =====

integer RBankSelect(bits(5) mode, integer usr, integer fiq, integer irq,
    integer svc, integer abt, integer und, integer hyp)

integer result;
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

return result;
```

## Library pseudocode for aarch32/functions/registers/Rmode

```
// Rmode[] - non-assignment form
// =====

bits(32) Rmode[integer n, bits(5) mode]
    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;
    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>;
    else
        return _R[LookUpRIndex(n, mode)]<31:0>;

// Rmode[] - assignment form
// =====

Rmode[integer n, bits(5) mode] = bits(32) value
    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;
    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;
    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(value, 64);
        else
            _R[LookUpRIndex(n, mode)]<31:0> = value;

    return;
```

## Library pseudocode for aarch32/functions/registers/S

```
// S[] - non-assignment form
// =====

bits(32) S[integer n]
    assert n >= 0 && n <= 31;
    constant bits(128) vreg = V[n DIV 4, 128];
    return Elem[vreg, n MOD 4, 32];

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    bits(128) vreg = V[n DIV 4, 128];
    Elem[vreg, n MOD 4, 32] = value;
    V[n DIV 4, 128] = vreg;
    return;
```

## 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.

array bits(64) _Dclone[0..31];
```

## Library pseudocode for aarch32/functions/system/AArch32.ExceptionReturn

```
// AArch32.ExceptionReturn()
// =====

AArch32.ExceptionReturn(bits(32) new_pc_in, bits(32) spsr)
    bits(32) new_pc = new_pc_in;
    SynchronizeContext();
    // Attempts to change to an illegal mode or state will invoke the Illegal Execution state
    // mechanism
    SetPSTATEFromPSR(spsr);
    ClearExclusiveLocal(ProcessorID());
    SendEventLocal();

    if PSTATE.IL == '1' then
        // If the exception return is illegal, PC[1:0] are UNKNOWN
        new_pc<1:0> = bits(2) UNKNOWN;
    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

    constant boolean branch_conditional = AArch32.CurrentCond() != '111x';
    BranchTo(new_pc, BranchType ERET, branch_conditional);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

## Library pseudocode for aarch32/functions/system/AArch32.ExecutingCP10or11Instr

```
// AArch32.ExecutingCP10or11Instr()
// =====

boolean AArch32.ExecutingCP10or11Instr()
    instr = ThisInstr();
    instr_set = 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');
```

## Library pseudocode for aarch32/functions/system/AArch32.ITAdvance

```
// AArch32.ITAdvance()
// =====

AArch32.ITAdvance()
    if PSTATE.IT<2:0> == '000' then
        PSTATE.IT = '00000000';
    else
        PSTATE.IT<4:0> = LSL(PSTATE.IT<4:0>, 1);
    return;
```

### 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].

AArch32.SysRegRead(integer cp_num, bits(32) instr, integer t);
```

### 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].

AArch32.SysRegRead64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### Library pseudocode for aarch32/functions/system/AArch32.SysRegReadCanWriteAPSR

```
// AArch32.SysRegReadCanWriteAPSR()
// =====
// Determines whether the AArch32 System register read instruction can write to APSR flags.

boolean AArch32.SysRegReadCanWriteAPSR(integer cp_num, bits(32) instr)
    assert UsingAArch32();
    assert (cp_num IN {14,15});
    assert cp_num == UInt(instr<11:8>);

    opc1 = UInt(instr<23:21>);
    opc2 = UInt(instr<7:5>);
    CRn = UInt(instr<19:16>);
    CRm = UInt(instr<3:0>);

    if cp_num == 14 && opc1 == 0 && CRn == 0 && CRm == 1 && opc2 == 0 then // DBGDSCRint
        return TRUE;

    return FALSE;
```

### 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.

AArch32.SysRegWrite(integer cp_num, bits(32) instr, integer t);
```

### 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.

AArch32.SysRegWrite64(integer cp_num, bits(32) instr, integer t, integer t2);
```

### 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.

AArch32.SysRegWriteM(integer cp_num, bits(32) instr, bits(32) address);
```

## 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.

AArch32.WriteMode(bits(5) mode)
    (valid,el) = 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;
```

## 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.

AArch32.WriteModeByInstr(bits(5) mode)
    (valid,el) = 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;

    // * A change to or from Hyp mode.
    if (PSTATE.M == M32\_Hyp || mode == M32\_Hyp) && PSTATE.M != mode then
        valid = FALSE;

    // * 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;

    if !valid then
        PSTATE.IL = '1';
    else
        AArch32.WriteMode(mode);
```



## Library pseudocode for aarch32/functions/system/BadMode

```
// BadMode()
// =====

boolean BadMode(bits(5) mode)
// Return TRUE if 'mode' encodes a mode that is not valid for this implementation
boolean valid;
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
return !valid;
```

## 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.

BankedRegisterAccessValid(bits(5) SYSm, bits(5) mode)

case SYSm of
    when '000xx', '00100' // R8_usr to R12_usr
        if mode != M32\_FIQ then UNPREDICTABLE;
    when '00101' // SP_usr
        if mode == M32\_System then UNPREDICTABLE;
    when '00110' // LR_usr
        if mode IN {M32\_Hyp, M32\_System} then UNPREDICTABLE;
    when '010xx', '0110x', '01110' // R8_fiq to R12_fiq, SP_fiq, LR_fiq
        if mode == M32\_FIQ then UNPREDICTABLE;
    when '1000x' // LR_irq, SP_irq
        if mode == M32\_IRQ then UNPREDICTABLE;
    when '1001x' // LR_svc, SP_svc
        if mode == M32\_Svc then UNPREDICTABLE;
    when '1010x' // LR_abt, SP_abt
        if mode == M32\_Abort then UNPREDICTABLE;
    when '1011x' // LR_und, SP_und
        if mode == M32\_Undef then UNPREDICTABLE;
    when '1110x' // LR_mon, SP_mon
        if (!HaveEL\(EL3\) || CurrentSecurityState\(\) != SS\_Secure ||
            mode == M32\_Monitor) then UNPREDICTABLE;
    when '11110' // ELR_hyp, only from Monitor or Hyp mode
        if !HaveEL\(EL2\) || ! mode IN {M32\_Monitor, M32\_Hyp} then UNPREDICTABLE;
    when '11111' // SP_hyp, only from Monitor mode
        if !HaveEL\(EL2\) || mode != M32\_Monitor then UNPREDICTABLE;
    otherwise
        UNPREDICTABLE;

return;
```

## Library pseudocode for aarch32/functions/system/CPSRWriteByInstr

```
// CPSRWriteByInstr()
// =====
// Update PSTATE.<N,Z,C,V,Q,GE,E,A,I,F,M> from a CPSR value written by an MSR instruction.

CPSRWriteByInstr(bits(32) value, bits(4) bytemask)
    privileged = PSTATE.EL != EL0; // PSTATE.<A,I,F,M> are not writable at EL0

    // 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

    if bytemask<2> == '1' then
        if IsFeatureImplemented(FEAT_SSBS) then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if IsFeatureImplemented(FEAT_DIT) then
            PSTATE.DIT = value<21>;
        // Bit <20> is RES0
        PSTATE.GE = value<19:16>;

    if bytemask<1> == '1' then
        // Bits <15:10> are RES0
        PSTATE.E = value<9>; // PSTATE.E is writable at EL0
        if privileged then
            PSTATE.A = value<8>;

    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>);

    return;
```

## Library pseudocode for aarch32/functions/system/ConditionPassed

```
// ConditionPassed()
// =====

boolean ConditionPassed()
    return ConditionHolds(AArch32.CurrentCond());
```

## Library pseudocode for aarch32/functions/system/CurrentCond

```
// CurrentCond()
// =====

bits(4) AArch32.CurrentCond();
```

## Library pseudocode for aarch32/functions/system/InITBlock

```
// InITBlock()
// =====

boolean InITBlock()
    if CurrentInstrSet() == InstrSet\_T32 then
        return PSTATE.IT<3:0> != '0000';
    else
        return FALSE;
```

## Library pseudocode for aarch32/functions/system/LastInITBlock

```
// LastInITBlock()
// =====

boolean LastInITBlock()
    return (PSTATE.IT<3:0> == '1000');
```

## Library pseudocode for aarch32/functions/system/SPSRWriteByInstr

```
// SPSRWriteByInstr()
// =====

SPSRWriteByInstr(bits(32) value, bits(4) bytemask)

    bits(32) new_spsr = 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

    if bytemask<2> == '1' then
        new_spsr<23:16> = value<23:16>; // IL bit, GE[3:0] flags

    if bytemask<1> == '1' then
        new_spsr<15:8> = value<15:8>; // IT[7:2] bits, E bit, A interrupt mask

    if bytemask<0> == '1' then
        new_spsr<7:0> = value<7:0>; // I,F interrupt masks, T bit, Mode bits

    SPSR\_curr[] = new_spsr; // UNPREDICTABLE if User or System mode

    return;
```

## Library pseudocode for aarch32/functions/system/SPSRAccessValid

```
// SPSRAccessValid()
// =====
// Checks for MRS (Banked register) or MSR (Banked register) accesses to the SPSRs
// that are UNPREDICTABLE

SPSRAccessValid(bits(5) SYSm, bits(5) mode)
    case SYSm of
        when '01110' // SPSR_fiq
            if mode == M32\_FIQ then UNPREDICTABLE;
        when '10000' // SPSR_irq
            if mode == M32\_IRQ then UNPREDICTABLE;
        when '10010' // SPSR_svc
            if mode == M32\_Svc then UNPREDICTABLE;
        when '10100' // SPSR_abt
            if mode == M32\_Abort then UNPREDICTABLE;
        when '10110' // SPSR_und
            if mode == M32\_Undef then UNPREDICTABLE;
        when '11100' // SPSR_mon
            if (!HaveEL(EL3) || mode == M32\_Monitor ||
                CurrentSecurityState() != SS\_Secure) then UNPREDICTABLE;
        when '11110' // SPSR_hyp
            if !HaveEL(EL2) || mode != M32\_Monitor then UNPREDICTABLE;
        otherwise
            UNPREDICTABLE;

    return;
```

## Library pseudocode for aarch32/functions/system/SelectInstrSet

```
// SelectInstrSet()
// =====

SelectInstrSet(InstrSet iset)
    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;
```

## 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.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast,
    TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op          = TLBIOp\_DALL;
    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);
    return;
```

## 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.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.DTLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.ITLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast,
    TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    r.op = TLBIOp\_IALL;
    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);
    return;
```

## 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.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2};
    assert security == SS\_NonSecure;

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```



## 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.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(32) Rt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBIMemAttr attr)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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);
    return;
```

## Library pseudocode for aarch32/functions/v6simd/Sat

```
// Sat()
// =====

bits(N) Sat(integer i, integer N, boolean unsigned)
    result = if unsigned then UnsignedSat(i, N) else SignedSat(i, N);
    return result;
```

## Library pseudocode for aarch32/functions/v6simd/SignedSat

```
// SignedSat()
// =====

bits(N) SignedSat(integer i, integer N)
    (result, -) = SignedSatQ(i, N);
    return result;
```

## Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSatQ(i, N);
    return result;
```

## Library pseudocode for aarch32/ic/AArch32.IC

```
// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(CacheOpScope opscope)
    regval = bits(32) UNKNOWN;
    AArch32.IC(regval, opscope);

// AArch32.IC()
// =====
// Perform Instruction Cache Operation.

AArch32.IC(bits(32) regval, CacheOpScope opscope)
    CacheRecord cache;

    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 then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval, 64);
        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;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_asid_valid = FALSE;

        need_translate = ICInstNeedsTranslation(opscope);

        cache.shareability = Shareability_NSH;
        cache.vaddress = ZeroExtend(regval, 64);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        constant integer size = 0;
        constant boolean aligned = TRUE;
        constant AccessDescriptor accdesc = CreateAccDescIC(cache);
        constant AddressDescriptor memaddrdesc = AArch32.TranslateAddress(regval, accdesc, aligned,
                                                                           size);

        if IsFault(memaddrdesc) then
            AArch32.Abort(memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;
```

## Library pseudocode for aarch32/predictionrestrict/AArch32.RestrictPrediction

```
// AArch32.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = 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();

    constant bit ns  = val<26>;
    constant bit nse = bit UNKNOWN;
    ss = TargetSecurityState(ns, nse);

    c.security = ss;
    c.target_el = target_el;

    if EL2Enabled() then
        if PSTATE.EL IN {EL0, EL1} then
            c.is_vmid_valid = TRUE;
            c.all_vmid      = FALSE;
            c.vmid          = VMID[];

            elseif target_el IN {EL0, EL1} then
                c.is_vmid_valid = TRUE;
                c.all_vmid      = val<27> == '1';
                c.vmid          = ZeroExtend(val<23:16>, 16);           // Only valid if val<27> == '0';
            else
                c.is_vmid_valid = FALSE;
        else
            c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elseif target_el == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = val<8> == '1';
        c.asid          = ZeroExtend(val<7:0>, 16);           // Only valid if val<8> == '0';

    else
        c.is_asid_valid = FALSE;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```



```

// AArch32.DefaultTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, without TEX remap

MemoryAttributes AArch32.DefaultTEXDecode(bits(3) TEX_in, bit c_in, bit b_in, bit s)
    MemoryAttributes memattrs;
    bits(3) TEX = TEX_in;
    bit c = c_in;
    bit b = b_in;

    // Reserved values map to allocated values
    if (TEX == '001' && c:b == '01') || (TEX == '010' && c:b != '00') || TEX == '011' then
        bits(5) texcb;
        (-, texcb) = ConstrainUnpredictableBits(Unpredictable\_RESTEXCB, 5);
        TEX = texcb<4:2>; c = texcb<1>; b = texcb<0>;

    // 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 = MemoryAttributes 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;
    otherwise
        // Reserved, handled above
        Unreachable();

    // 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';

    return memattrs;

```

### Library pseudocode for aarch32/translation/attrs/AArch32.MAIRAttr

```

// AArch32.MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) AArch32.MAIRAttr(integer index, MAIRType mair)
    assert (index < 8);
    return Elem[mair, index, 8];

```

## Library pseudocode for aarch32/translation/attrs/AArch32.RemappedTEXDecode

```
// AArch32.RemappedTEXDecode()
// =====
// Apply short-descriptor format memory region attributes, with TEX remap

MemoryAttributes AArch32.RemappedTEXDecode(Regime regime, bits(3) TEX, bit c, bit b, bit s)

MemoryAttributes memattrs;
PRRR_Type prrr;
NMRR_Type nmrr;

region = UInt(TEX<0>:c:b);          // TEX<2:1> are ignored in this mapping scheme
if region == 6 then
    return MemoryAttributes IMPLEMENTATION_DEFINED;

if regime == Regime\_EL30 then
    prrr = PRRR_S;
    nmrr = NMRR_S;
elseif HaveEL(EL3) && ELUsingAArch32(EL3) then
    prrr = PRRR_NS;
    nmrr = NMRR_NS;
else
    prrr = PRRR;
    nmrr = NMRR;

constant integer base = 2 * region;
attrfield = Elem[prrr, region, 2];

if attrfield == '11' then          // Reserved, maps to allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable\_RESPPRR, 2);

case attrfield of
    when '00'                      // Device-nGnRnE
        memattrs.memtype          = MemType\_Device;
        memattrs.device           = DeviceType\_nGnRnE;
        memattrs.shareability     = Shareability\_OSH;
    when '01'                      // Device-nGnRE
        memattrs.memtype          = MemType\_Device;
        memattrs.device           = DeviceType\_nGnRE;
        memattrs.shareability     = Shareability\_OSH;
    when '10'
        NSn = if s == '0' then prrr.NS0 else prrr.NS1;
        NOSm = prrr<region+24> AND NSn;
        IRn = nmrr<base+1:base>;
        ORn = nmrr<base+17:base+16>;

        memattrs.memtype = MemType\_Normal;
        memattrs.inner    = DecodeSDFAttr(IRn);
        memattrs.outer    = DecodeSDFAttr(ORn);
        if memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC then
            memattrs.shareability = Shareability\_OSH;
        else
            constant bits(2) sh = NSn:NOSm;
            memattrs.shareability = DecodeShareability(sh);
    when '11'
        Unreachable();

// 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';

return memattrs;
```



## 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.

FaultRecord AArch32.CheckBreakpoint(FaultRecord fault_in, bits(32) vaddress,
                                     AccessDescriptor accdesc, integer size)

    assert ELUsingAArch32(S1TranslationRegime()) ;
    assert size IN {2,4};

    FaultRecord fault = fault_in;
    match = FALSE;
    mismatch = FALSE;
    BreakpointInfo brkptinfo;

    for i = 0 to NumBreakpointsImplemented() - 1
        brkptinfo = AArch32.BreakpointMatch(i, vaddress, accdesc, size);
        match = match || brkptinfo.match;
        mismatch = mismatch || brkptinfo.mismatch;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elseif (match || mismatch) then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Breakpoint;
        fault.vaddress = ZeroExtend(vaddress, 64);

    return fault;
```

## 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.

FaultRecord AArch32.CheckDebug(bits(32) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc, ZeroExtend(vaddress, 64));

    constant boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType\_DC);
    constant boolean i_side = (accdesc.acctype == AccessType\_IFETCH);
    generate_exception = AArch32.GenerateDebugExceptions() && DBGDSCRext.MDBGen == '1';
    halt = HaltOnBreakpointOrWatchpoint();
    // Relative priority of Vector Catch and Breakpoint exceptions not defined in the architecture
    vector_catch_first = ConstrainUnpredictableBool(Unpredictable\_BPVECTORCATCHPRI);

    if i_side && vector_catch_first && generate_exception then
        fault = AArch32.CheckVectorCatch(fault, vaddress, size);

    if fault.statuscode == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(fault, vaddress, accdesc, size);
        elseif i_side then
            fault = AArch32.CheckBreakpoint(fault, vaddress, accdesc, size);

    if fault.statuscode == Fault\_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(fault, vaddress, size);

    return fault;
```

## 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.

FaultRecord AArch32.CheckVectorCatch(FaultRecord fault_in, bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    FaultRecord fault = fault_in;
    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable\_VCMATCHHALF);

    if match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_VectorCatch;

    return fault;
```

## 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.

FaultRecord AArch32.CheckWatchpoint(FaultRecord fault_in, bits(32) vaddress,
                                     AccessDescriptor accdesc, integer size)
    assert ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault = fault_in;
    boolean match = FALSE;

    if accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop != CacheOp\_Invalidate then
            return fault;
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return fault;
    elsif IsDataAccess(accdesc.acctype) then
        return fault;

    for i = 0 to NumWatchpointsImplemented() - 1
        constant WatchpointInfo watchptinfo = AArch32.WatchpointMatch(i, vaddress, size, accdesc);
        match = match || watchptinfo.value_match;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt\_Watchpoint;
        EDWAR = ZeroExtend(vaddress, 64);
        Halt(reason);
    elsif match then
        fault.statuscode = Fault\_Debug;
        fault.debugmoe = DebugException\_Watchpoint;
        fault.vaddress = ZeroExtend(vaddress, 64);

    return fault;
```

## Library pseudocode for aarch32/translation/faults/AArch32.IPAIsOutOfRange

```
// AArch32.IPAIsOutOfRange()
// =====
// Check intermediate physical address bits not resolved by translation are ZERO

boolean AArch32.IPAIsOutOfRange(S2TTWParams walkparams, bits(40) ipa)
    // Input Address size
    constant iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

## Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFaultDueToMemType

```
// AArch32.S1HasAlignmentFaultDueToMemType()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses to memory type

boolean AArch32.S1HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
                                                bit ntlsmd, MemoryAttributes memattrs)

    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 !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

## 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

boolean AArch32.S1LDHasPermissionsFault(Regime regime, S1TTWParams walkparams, Permissions perms,
                                         MemType memtype, PASpace paspace, AccessDescriptor accdesc)

    bit r, w, x;
    bit pr, pw;
    bit ur, uw;
    bit xn;
    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

        // 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

        xn = perms.xn OR perms.xn_table;
        pxn = perms.pxn OR perms.pxn_table;

        ux = ur AND NOT(xn OR (uw AND walkparams.wxn));
        px = pr AND NOT(xn OR pxn OR (pw AND walkparams.wxn) OR (uw AND walkparams.uwxn));

        if IsFeatureImplemented(FEAT_PAN) && accdesc.pan then
            pan = PSTATE.PAN AND (ur OR uw);
            pr = pr AND NOT(pan);
            pw = pw AND NOT(pan);

        (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);
    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

        // 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

        xn = perms.xn OR perms.xn_table;
        x = NOT(xn OR (w AND walkparams.wxn));

    if accdesc.acctype == AccessType\_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable INSTRDEVICE);
        if constraint == Constraint\_FAULT && memtype == MemType\_Device then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

## 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

boolean AArch32.S1SDHasPermissionsFault(Regime regime, Permissions perms_in, MemType memtype,
                                         PASpace paspace, AccessDescriptor accdesc)

    Permissions perms = perms_in;
    bit pr, pw;
    bit ur, uw;
    SCTLR_Type sctlr;
    if regime == Regime\_EL30 then
        sctlr = SCTLR_S;
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        sctlr = SCTLR_NS;
    else
        sctlr = SCTLR;

    if sctlr.AFE == '0' then
        // Map Reserved encoding '100'
        if perms.ap == '100' then
            perms.ap = bits(3) IMPLEMENTATION_DEFINED "Reserved short descriptor AP encoding";

        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
        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

    ux = ur AND NOT(perms.xn OR (uw AND sctlr.WXN));
    px = 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
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = 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);

    if accdesc.acctype == AccessType\_IFETCH then
        if (memtype == MemType\_Device &&
            ConstrainUnpredictable(Unpredictable\_INSTRDEVICE) == Constraint\_FAULT) then
            return TRUE;
        else
            return x == '0';
    elsif accdesc.acctype IN {AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elsif accdesc.write then
        return w == '0';
    else
        return r == '0';
```

## 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

boolean AArch32.S2HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
MemoryAttributes memattrs)

    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elsif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elsif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(S2TTWParams walkparams, Permissions perms, MemType memtype,
AccessDescriptor accdesc)

    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    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');

        x = if accdesc.el == EL0 then ux else px;
    else
        x = r AND NOT(perms.s2xn);

    if accdesc.acctype == AccessType\_TTW then
        return (walkparams.ptw == '1' && memtype == MemType\_Device) || r == '0';

    elsif accdesc.acctype == AccessType\_IFETCH then
        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';
```

## Library pseudocode for aarch32/translation/faults/AArch32.S2InconsistentSL

```
// AArch32.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 T0SZ and SL fields

boolean AArch32.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch32.S2StartLevel(walkparams.sl0);
    levels     = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride     = granulebits - 3;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        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 accomodate 1 more stride in the level + concatenation of up to 2^4 tables
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize        = AArch32.S2IASize(walkparams.t0sz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

## Library pseudocode for aarch32/translation/faults/AArch32.VAIsOutOfRange

```
// AArch32.VAIsOutOfRange()
// =====
// Check virtual address bits not resolved by translation are identical
// and of accepted value

boolean AArch32.VAIsOutOfRange(Regime regime, S1TTWParams walkparams, bits(32) va)
    if regime == Regime\_EL2 then
        // Input Address size
        constant iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elsif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        constant lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        constant up_iasize = AArch32.S1IASize(walkparams.t1sz);
        return !IsZero(va<31:lo_iasize>) && !IsOnes(va<31:up_iasize>);
    else
        return FALSE;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS1TLBContext

```
// AArch32.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch32.GetS1TLBContext(Regime regime, SecurityState ss, bits(32) va)
    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);

    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;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.GetS2TLBContext

```
// AArch32.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch32.GetS2TLBContext(FullAddress ipa)
    assert ipa.paspace == PAS_NonSecure;

    TLBContext tlbcontext;

    tlbcontext.ss          = SS_NonSecure;
    tlbcontext.regime      = Regime_EL10;
    tlbcontext.ipaspace    = ipa.paspace;
    tlbcontext.vmid        = ZeroExtend(VTTBR.VMID, 16);
    tlbcontext.tg          = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia          = ZeroExtend(ipa.address, 64);
    tlbcontext.cnp         = if IsFeatureImplemented(FEAT_TTCNP) then VTTBR.CnP else '0';

    return tlbcontext;
```



## 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

TLBContext AArch32.TLBContextEL10(SecurityState ss, bits(32) va)
    TLBContext tlbcontext;
    TTBCR_Type ttbcr;
    TTBR0_Type ttbr0;
    TTBR1_Type ttbr1;
    CONTEXTIDR_Type contextidr;

    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;

    tlbcontext.ss = ss;
    tlbcontext.regime = Regime\_EL10;

    if AArch32.EL2Enabled(ss) then
        tlbcontext.vmid = ZeroExtend(VTTBR.VMID, 16);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID, 16);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && ttbcr.EAE == '1' then
        if AArch32.GetVARange(va, ttbcr.T0SZ, ttbcr.T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = ttbr0.CnP;
        else
            tlbcontext.cnp = ttbr1.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/tlbcontext/AArch32.TLBContextEL2

```
// AArch32.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch32.TLBContextEL2(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss = SS\_NonSecure;
    tlbcontext.regime = Regime\_EL2;
    tlbcontext.ia = ZeroExtend(va, 64);
    tlbcontext.tg = TGx\_4KB;
    tlbcontext.cnp = if IsFeatureImplemented(FEAT_TTCNP) then HTTBR.CnP else '0';

    return tlbcontext;
```

## 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

TLBContext AArch32.TLBContextEL30(bits(32) va)
    TLBContext tlbcontext;

    tlbcontext.ss      = SS\_Secure;
    tlbcontext.regime = Regime\_EL30;

    if TTBCR_S.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if TTBCR_S.A1 == '0' then TTBR0_S.ASID else TTBR1_S.ASID, 16);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID, 16);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va, 64);

    if IsFeatureImplemented(FEAT_TTCNP) && TTBCR_S.EAE == '1' then
        if AArch32.GetVARange(va, TTBCR_S.T0SZ, TTBCR_S.T1SZ) == VARange\_LOWER then
            tlbcontext.cnp = TTBR0_S.CnP;
        else
            tlbcontext.cnp = TTBR1_S.CnP;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch32/translation/translation/AArch32.EL2Enabled

```
// AArch32.EL2Enabled()
// =====
// Returns whether EL2 is enabled for the given Security State

boolean AArch32.EL2Enabled(SecurityState ss)
    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 boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
    else
        return HaveEL(EL2);
```

## Library pseudocode for aarch32/translation/translation/AArch32.FullTranslate

```
// AArch32.FullTranslate()
// =====
// Perform address translation as specified by VMSA-A32

AddressDescriptor AArch32.FullTranslate(bits(32) va, AccessDescriptor accdesc, boolean aligned)

    // Prepare fault fields in case a fault is detected
    FaultRecord fault = NoFault(accdesc, ZeroExtend(va, 64));
    constant Regime regime = TranslationRegime(accdesc.el);

    // First Stage Translation
    AddressDescriptor ipa;
    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);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);

    if regime == Regime\_EL10 && EL2Enabled() then
        ipa.vaddress = ZeroExtend(va, 64);
        AddressDescriptor pa;
        (fault, pa) = AArch32.S2Translate(fault, ipa, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(ZeroExtend(va, 64), fault);
        else
            return pa;
    else
        return ipa;
```

## Library pseudocode for aarch32/translation/translation/AArch32.OutputDomain

```
// AArch32.OutputDomain()
// =====
// Determine the domain the translated output address

bits(2) AArch32.OutputDomain(Regime regime, bits(4) domain)
    bits(2) Dn;
    if regime == Regime\_EL30 then
        Dn = Elem[DACR_S, UInt(domain), 2];
    elsif HaveEL(EL3) && ELUsingAArch32(EL3) then
        Dn = Elem[DACR_NS, UInt(domain), 2];
    else
        Dn = Elem[DACR, UInt(domain), 2];

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable\_RESDACR, 2);

    return Dn;
```



```

// AArch32.S1DisabledOutput()
// =====
// Flat map the VA to IPA/PA, depending on the regime, assigning default memory attributes

(FaultRecord, AddressDescriptor) AArch32.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                         bits(32) va, boolean aligned,
                                                         AccessDescriptor accdesc)

FaultRecord fault = fault_in;
// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

MemoryAttributes memattrs;
bit default_cacheable;
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;
else
    default_cacheable = '0';

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;
    memattrs.xs = '0';
elseif 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;
    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';

bit ntlsmd;
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;
    else
        ntlsmd = '1';

```

```

if AArch32.S1HasAlignmentFaultDueToMemType(accdesc, aligned, ntlsmd, memattrs) then
    fault.statuscode = Fault\_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va, 56);
oa.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);

```

## Library pseudocode for aarch32/translation/translation/AArch32.S1Enabled

```

// AArch32.S1Enabled()
// =====
// Returns whether stage 1 translation is enabled for the active translation regime

boolean AArch32.S1Enabled(Regime regime, SecurityState ss)
    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';
    else
        return EL2Enabled() && HCR_EL2.<TGE,DC> == '00' && SCTLR.M == '1';

```

## 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

(FaultRecord, AddressDescriptor) AArch32.S1TranslateLD(FaultRecord fault_in, Regime regime,
                                                    bits(32) va, boolean aligned,
                                                    AccessDescriptor accdesc)

FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    return AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);

walkparams = AArch32.GetS1TTWParams(regime, va);

if AArch32.VAIsOutOfRange(regime, walkparams, va) then
    fault.level = 1;
    fault.statuscode = Fault\_Translation;
    return (fault, AddressDescriptor UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkLD(fault, regime, walkparams, accdesc, va);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSEA 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;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
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;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = StageOA(ZeroExtend(va, 64), walkparams.dl28, walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa);
```





```

// AArch32.S1TranslateSD()
// =====
// Perform a stage 1 translation using short-descriptor format mapping VA to IPA/PA
// depending on the regime

(FaultRecord, AddressDescriptor, SDType) AArch32.S1TranslateSD(FaultRecord fault_in, Regime regime,
                                                             bits(32) va, boolean aligned,
                                                             AccessDescriptor accdesc)

FaultRecord fault = fault_in;

if !AArch32.S1Enabled(regime, accdesc.ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, va, aligned, accdesc);
    return (fault, ipa, SDType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, accdesc, va);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN, SDType UNKNOWN);

domain = AArch32.OutputDomain(regime, walkstate.domain);
SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

bit ntlsmd;
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;
    else
        ntlsmd = '1';

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;

if fault.statuscode != Fault\_None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
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;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;

```

```
else
    memattrs.shareability = EffectiveShareability(memattrs);

// Output Address
oa = AArch32.SDStageOA(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va, 64), oa, memattrs);

return (fault, ipa, walkstate.sdftype);
```

## Library pseudocode for aarch32/translation/translation/AArch32.S2Translate

```
// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean aligned, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<55:40>);

    if !ELStateUsingAArch32(EL2, accdesc.ss == SS_Secure) then
        slaarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, slaarch64, aligned, accdesc);

    // 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;

    walkparams = AArch32.GetS2TTWParams();

    if walkparams.vm == '0' then
        // Stage 2 is disabled
        return (fault, ipa);

    if AArch32.IPAIsOutOfRange(walkparams, ipa.paddress.address<39:0>) then
        fault.statuscode = Fault_Translation;
        fault.level = 1;
        return (fault, AddressDescriptor UNKNOWN);

    TTWState walkstate;
    (fault, walkstate) = AArch32.S2Walk(fault, walkparams, accdesc, ipa);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch32.S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elseif AArch32.S2HasPermissionsFault(walkparams,
                                          walkstate.permissions,
                                          walkstate.memattrs.memtype,
                                          accdesc) then
        fault.statuscode = Fault_Permission;
    MemoryAttributes s2_memattrs;
    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;

    s2aarch64 = FALSE;
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);
    ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
    // Output Address
    oa = StageOA(ipa_64, walkparams.d128, walkparams.tgx, walkstate);
    pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);

    return (fault, pa);
```

## 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

FullAddress AArch32.SDStageOA(FullAddress baseaddress, bits(32) va, SDFType sdftype)
    constant integer tsize = SDFSize(sdftype);

    // Output Address
    FullAddress oa;
    oa.address = baseaddress.address<55:tsize> : va<tsize-1:0>;
    oa.paspace = baseaddress.paspace;
    return oa;
```

## Library pseudocode for aarch32/translation/translation/AArch32.TranslateAddress

```
// AArch32.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch32.TranslateAddress(bits(32) va, AccessDescriptor accdesc,
    boolean aligned, integer size)

    constant Regime regime = TranslationRegime(PSTATE.EL);
    if !RegimeUsingAArch32(regime) then
        return AArch64.TranslateAddress(ZeroExtend(va, 64), accdesc, aligned, size);

    AddressDescriptor result = AArch32.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) then
        result.fault = AArch32.CheckDebug(va, accdesc, size);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

## Library pseudocode for aarch32/translation/translation/SDFSize

```
// SDFSize()
// =====
// Returns the short-descriptor format translation granule size

AddressSize SDFSize(SDFType sdftype)
    case sdftype of
        when SDFType_SmallPage      return 12;
        when SDFType_LargePage      return 16;
        when SDFType_Section        return 20;
        when SDFType_Supersection   return 24;
        otherwise Unreachable();
```

## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeLD

```
// AArch32.DecodeDescriptorTypeLD()
// =====
// Determine whether the long-descriptor is a page, block or table

DescriptorType AArch32.DecodeDescriptorTypeLD(bits(64) descriptor, integer level)
    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;
```

## Library pseudocode for aarch32/translation/walk/AArch32.DecodeDescriptorTypeSD

```
// AArch32.DecodeDescriptorTypeSD()
// =====
// Determine the type of the short-descriptor

SDFType AArch32.DecodeDescriptorTypeSD(bits(32) descriptor, integer level)
    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;
```

## Library pseudocode for aarch32/translation/walk/AArch32.S1IASize

```
// AArch32.S1IASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

AddressSize AArch32.S1IASize(bits(3) txsz)
    return 32 - UInt(txsz);
```



```

// AArch32.S1WalkLD()
// =====
// Traverse stage 1 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkLD(FaultRecord fault_in, Regime regime,
                                         S1TTWParams walkparams, AccessDescriptor accdesc,
                                         bits(32) va)

FaultRecord fault = fault_in;
bits(3) txsz;
bits(64) ttbr;
bit epd;
VARange varange;
if regime == Regime EL2 then
    ttbr = HTTBR;
    txsz = walkparams.t0sz;
    varange = VARange LOWER;
else
    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    bits(64) ttbr0;
    bits(64) ttbr1;
    TTBCR_Type ttbcr;
    if regime == Regime EL30 then
        ttbcr = TTBCR_S;
        ttbr0 = TTBR0_S;
        ttbr1 = TTBR1_S;
    elseif HaveEL(EL3) && ELUsingAArch32(EL3) then
        ttbcr = TTBCR_NS;
        ttbr0 = TTBR0_NS;
        ttbr1 = TTBR1_NS;
    else
        ttbcr = TTBCR;
        ttbr0 = TTBR0;
        ttbr1 = TTBR1;

    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;

if regime != Regime EL2 && epd == '1' then
    fault.level = 1;
    fault.statuscode = Fault Translation;
    return (fault, TTWState UNKNOWN);

// Input Address size
iasize = AArch32.S1IASize(txsz);
granulebits = TGxGranuleBits(walkparams.tgx);
stride = granulebits - 3;
startlevel = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
levels = FINAL\_LEVEL - startlevel;

if !IsZero(ttbr<47:40>) then
    fault.statuscode = Fault AddressSize;
    fault.level = 0;
    return (fault, TTWState UNKNOWN);

FullAddress baseaddress;
constant baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspac = if accdesc.ss == SS Secure then PAS Secure else PAS NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>:Zeros(baselsb), 56);

TTWState walkstate;
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';

bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

DescriptorType desctype;
integer msb_residual = iasize - 1;
repeat
    fault.level = walkstate.level;
    constant indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
    constant indexmsb = msb_residual;
    constant bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000', 40);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    constant boolean toplevel = walkstate.level == startlevel;
    constant AccessDescriptor walkaccess = 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
        s2aligned = TRUE;
        (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, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                            fault, 64);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                            fault, 64);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

    case desctype of
        when DescriptorType\_Table
            if !IsZero(descriptor<47:40>) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, TTWState UNKNOWN);

    walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);

```



```

        if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<63> == '1' then
            walkstate.baseaddress.paspace = PAS\_NonSecure;

        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>);

        walkstate.level = walkstate.level + 1;
        msb_residual = indexlsb - 1;

        when DescriptorType Invalid
            fault.statuscode = Fault\_Translation;
            return (fault, TTWState UNKNOWN);

        when DescriptorType Leaf
            walkstate.istable = FALSE;

until desctype == DescriptorType\_Leaf;

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault\_AddressSize;
    return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault\_AccessFlag;
    return (fault, TTWState UNKNOWN);

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>;

constant indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;

memattr = descriptor<4:2>;
sh = descriptor<9:8>;
attr = AArch32.MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams);

return (fault, walkstate);

```



```

// AArch32.S1WalkSD()
// =====
// Traverse stage 1 translation tables in short format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S1WalkSD(FaultRecord fault_in, Regime regime,
                                         AccessDescriptor accdesc, bits(32) va)

FaultRecord fault = fault_in;
SCTLR_Type sctlr;
TTBCR_Type ttbcr;
TTBR0_Type ttbr0;
TTBR1_Type ttbr1;
// Determine correct translation control registers to use.
if regime == Regime EL30 then
    sctlr = SCTLR_S;
    ttbcr = TTBCR_S;
    ttbr0 = TTBR0_S;
    ttbr1 = TTBR1_S;
elseif 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;

assert ttbcr.EAE == '0';
ee = sctlr.EE;
afe = sctlr.AFE;
tre = sctlr.TRE;
constant integer ttbcr_n = UInt(ttbcr.N);
constant VARange varange = (if ttbcr_n == 0 || IsZero(va<31:(32-ttbcr_n)>) then VARange\_LOWER
                           else VARange\_UPPER);
constant integer n = if varange == VARange\_LOWER then ttbcr_n else 0;
bits(32) ttb;
bits(1) pd;
bits(2) irgn;
bits(2) rgn;
bits(1) s;
bits(1) nos;
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;

// 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, TTWState UNKNOWN);

FullAddress baseaddress;
baseaddress.paspace = if accdesc.ss == SS\_Secure then PAS\_Secure else PAS\_NonSecure;
baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n), 56);

constant integer startlevel = 1;
TTWState walkstate;
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;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va, 64);

if !AArch32.S1DCacheEnabled(regime) then
    walkaddress.memattrs = NormalNCMemAttr();
    walkaddress.memattrs.xs = walkstate.memattrs.xs;
else
    walkaddress.memattrs = walkstate.memattrs;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

bit nG;
bit ns;
bit pxn;
bits(3) ap;
bits(3) tex;
bit c;
bit b;
bit xn;
repeat
    fault.level = walkstate.level;

    bits(32) index;
    if walkstate.level == 1 then
        index = ZeroExtend(va<31-n:20>:'00', 32);
    else
        index = ZeroExtend(va<19:12>:'00', 32);

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index,
                                                                    56);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    constant boolean toplevel = walkstate.level == startlevel;
    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    if regime == Regime\_EL10 && AArch32.EL2Enabled(accdesc.ss) then
        s2aligned = TRUE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, s2aligned,
                                                    walkaccess);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, walkaccess, fault, 32);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, walkaccess, fault, 32);

    if fault.statuscode != Fault\_None then
        return (fault, TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

```

```

case walkstate.sdftype of
  when SDFTType\_Invalid
    fault.domain      = domain;
    fault.statuscode = Fault\_Translation;
    return (fault, TTWState\_UNKNOWN);

  when SDFTType\_Table
    domain = descriptor<8:5>;
    ns     = descriptor<3>;
    pxn    = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10),
                                              56);

    walkstate.level = 2;

  when SDFTType\_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(descriptor<31:12>:Zeros(12),
                                              56);

    walkstate.istable = FALSE;

  when SDFTType\_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(descriptor<31:16>:Zeros(16),
                                              56);

    walkstate.istable = FALSE;

  when SDFTType\_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(descriptor<31:20>:Zeros(20),
                                              56);

    walkstate.istable = FALSE;

  when SDFTType\_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(descriptor<8:5,23:20,31:24>:Zeros(24),

```

```

56);

walkstate.istable = FALSE;

until walkstate.sdftype != SDFTYPE\_Table;

if afe == '1' && ap<0> == '0' then
    fault.domain      = domain;
    fault.statuscode = Fault\_AccessFlag;
    return (fault, TTWState UNKNOWN);

// 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 = MemoryAttributes IMPLEMENTATION_DEFINED;

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;

return (fault, walkstate);

```

#### Library pseudocode for aarch32/translation/walk/AArch32.S2IASize

```

// AArch32.S2IASize()
// =====
// Retrieve the number of bits containing the input address for stage 2 translation

AddressSize AArch32.S2IASize(bits(4) t0sz)
    return 32 - SInt(t0sz);

```

#### Library pseudocode for aarch32/translation/walk/AArch32.S2StartLevel

```

// AArch32.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch32.S2StartLevel(bits(2) sl0)
    return 2 - UInt(sl0);

```



```

// AArch32.S2Walk()
// =====
// Traverse stage 2 translation tables in long format to obtain the final descriptor

(FaultRecord, TTWState) AArch32.S2Walk(FaultRecord fault_in, S2TTWParams walkparams,
                                       AccessDescriptor accdesc, AddressDescriptor ipa)

    FaultRecord fault = fault_in;

    if walkparams.sl0 == '1x' || AArch32.S2InconsistentSL(walkparams) then
        fault.statuscode = Fault\_Translation;
        fault.level      = 1;
        return (fault, TTWState UNKNOWN);

    // Input Address size
    iasize      = AArch32.S2IASize(walkparams.t0sz);
    startlevel  = AArch32.S2StartLevel(walkparams.sl0);
    levels      = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride      = granulebits - 3;

    if !IsZero(VTBR<47:40>) then
        fault.statuscode = Fault\_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    constant baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS\_NonSecure;
    baseaddress.address = ZeroExtend(VTBR<39:baselsb>:Zeros(baselsb), 56);

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    bits(64) descriptor;
    constant AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;

    if HCR2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    integer msb_residual = iasize - 1;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        constant indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
        constant indexmsb = msb_residual;
        constant bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000', 40);

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index, 56);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, 64);

        if fault.statuscode != Fault\_None then
            return (fault, TTWState UNKNOWN);

        desctype = AArch32.DecodeDescriptorTypeLD(descriptor, walkstate.level);

```



```

case desctype of
  when DescriptorType Table
    if !IsZero(descriptor<47:40>) then
      fault.statuscode = Fault AddressSize;
      return (fault, TTWState UNKNOWN);

      walkstate.baseaddress.address = ZeroExtend(descriptor<39:12>:Zeros(12), 56);
      walkstate.level = walkstate.level + 1;
      msb_residual = indexlsb - 1;

  when DescriptorType Invalid
    fault.statuscode = Fault Translation;
    return (fault, TTWState UNKNOWN);

  when DescriptorType Leaf
    walkstate.istable = FALSE;

until desctype == DescriptorType Leaf;

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
  fault.statuscode = Fault AddressSize;
  return (fault, TTWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
  fault.statuscode = Fault AccessFlag;
  return (fault, TTWState UNKNOWN);

// Unpack the descriptor into address and upper and lower block attributes
constant indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb), 56);

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';

memattr = descriptor<5:2>;
sh      = descriptor<9:8>;
s2aarch64 = FALSE;
walkstate.memattrs  = S2DecodeMemAttrs(memattr, sh, s2aarch64);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

## Library pseudocode for aarch32/translation/walk/RemapRegsHaveResetValues

```

// RemapRegsHaveResetValues()
// =====

boolean RemapRegsHaveResetValues();

```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS1TTWParams

```
// AArch32.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch32.GetS1TTWParams(Regime regime, bits(32) va)
    S1TTWParams walkparams;

    case regime of
        when Regime\_EL2 walkparams = AArch32.S1TTWParamsEL2();
        when Regime\_EL10 walkparams = AArch32.S1TTWParamsEL10(va);
        when Regime\_EL30 walkparams = AArch32.S1TTWParamsEL30(va);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetS2TTWParams

```
// AArch32.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch32.GetS2TTWParams()
    S2TTWParams walkparams;

    walkparams.tgx = TGx\_4KB;
    walkparams.s = VTCR.S;
    walkparams.t0sz = VTCR.T0SZ;
    walkparams.sl0 = VTCR.SL0;
    walkparams.irgn = VTCR.IRGN0;
    walkparams.orgn = VTCR.ORGNO;
    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(Unpredictable\_RESVTCRS, 4);

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.GetVARange

```
// AArch32.GetVARange()
// =====
// Select the translation base address for stage 1 long-descriptor walks

VARange AArch32.GetVARange(bits(32) va, bits(3) t0sz, bits(3) t1sz)
    // Lower range Input Address size
    constant AddressSize lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    constant AddressSize up_iasize = 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 VARange UNKNOWN;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1DCacheEnabled

```
// AArch32.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch32.S1DCacheEnabled(Regime regime)
    case regime of
        when Regime\_EL30 return SCTL_R_S.C == '1';
        when Regime\_EL2  return HSCTLR.C == '1';
        when Regime\_EL10
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTL_R_NS.C else SCTL_R.C) == '1';
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1ICacheEnabled

```
// AArch32.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch32.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime\_EL30 return SCTL_R_S.I == '1';
        when Regime\_EL2  return HSCTLR.I == '1';
        when Regime\_EL10
            return (if HaveEL(EL3) && ELUsingAArch32(EL3) then SCTL_R_NS.I else SCTL_R.I) == '1';
```

## 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).

S1TTWParams AArch32.S1TTWParamsEL10(bits(32) va)
    bits(64) mair;
    bit sif;
    TTBCR_Type ttbcr;
    TTBCR2_Type ttbcr2;
    SCTLR_Type sctlr;

    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';

    assert ttbcr.EAE == '1';
    S1TTWParams walkparams;

    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;

    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.ORG1;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then ttbcr.T2E AND ttbcr2.HPD1
            else '0');

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL2

```
// AArch32.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch32.S1TTWParamsEL2()
    S1TTWParams walkparams;

    walkparams.tgx = TGx\_4KB;
    walkparams.t0sz = HTCR.T0SZ;
    walkparams.irgn = HTCR.SH0;
    walkparams.orgn = HTCR.IRGN0;
    walkparams.sh = HTCR.ORGNO;
    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';

    walkparams.mair = HMAIR1:HMAIR0;

    return walkparams;
```

## Library pseudocode for aarch32/translation/walkparams/AArch32.S1TTWParamsEL30

```
// AArch32.S1TTWParamsEL30()
// =====
// Gather stage 1 translation table walk parameters for EL3&0 regime

S1TTWParams AArch32.S1TTWParamsEL30(bits(32) va)
    assert TTBCR_S.EAE == '1';
    S1TTWParams walkparams;

    walkparams.t0sz = TTBCR_S.T0SZ;
    walkparams.t1sz = TTBCR_S.T1SZ;
    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;

    varange = AArch32.GetVARange(va, walkparams.t0sz, walkparams.t1sz);
    if varange == VARange\_LOWER then
        walkparams.sh = TTBCR_S.SH0;
        walkparams.irgn = TTBCR_S.IRGN0;
        walkparams.orgn = TTBCR_S.ORGNO;
        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.ORGNO;
        walkparams.hpd = (if IsFeatureImplemented(FEAT_AA32HPD) then TTBCR_S.T2E AND TTBCR2_S.HPD1
            else '0');

    return walkparams;
```

## Library pseudocode for aarch64/debug/brbe/BRBCycleCountingEnabled

```
// BRBCycleCountingEnabled()
// =====
// Returns TRUE if the recording of cycle counts is allowed,
// FALSE otherwise.

boolean BRBCycleCountingEnabled()
    if HaveEL\(EL2\) && BRBCR_EL2.CC == '0' then return FALSE;
    if BRBCR_EL1.CC == '0' then return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/debug/brbe/BRBEBranch

```
// BRBEBranch()
// =====
// Called to write branch record for the following branches when BRB is active:
// direct branches,
// indirect branches,
// direct branches with link,
// indirect branches with link,
// returns from subroutines.

BRBEBranch(BranchType br_type, boolean cond, bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) && FilterBranchRecord(br_type, cond) then
        bits(6) branch_type;
        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();

        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bit lastfailed = (if IsFeatureImplemented(FEAT_TME) then BRBFCR_EL1.LASTFAILED
                                   else '0');
        constant bit transactional = (if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then '1'
                                       else '0');
        constant bits(2) el = PSTATE.EL;
        constant bit mispredict = (if BRBEMispredictAllowed() && BranchMispredict() then '1'
                                    else '0');

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                '11', PC64, target_address);

        BRBFCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

## Library pseudocode for aarch64/debug/brbe/BRBEBranchOnISB

```
// BRBEBranchOnISB()
// =====
// Returns TRUE if ISBs generate Branch records, and FALSE otherwise.

boolean BRBEBranchOnISB()
    return boolean IMPLEMENTATION_DEFINED "ISB generates Branch records";
```

## Library pseudocode for aarch64/debug/brbe/BRBEDebugStateExit

```
// BRBEDebugStateExit()
// =====
// Called to write Debug state exit branch record when BRB is active.

BRBEDebugStateExit(bits(64) target_address)
    if BranchRecordAllowed(PSTATE.EL) then
        // Debug state is a prohibited region, therefore ccu=1, cc=0, source_address=0
        constant bits(6) branch_type = '111001';
        constant bit ccu = '1';
        constant bits(14) cc = Zeros(14);
        constant bit lastfailed = (if IsFeatureImplemented(FEAT_TME) then BRBFCCR_EL1.LASTFAILED
                                   else '0');
        constant bit transactional = '0';
        constant bits(2) el = PSTATE.EL;
        constant bit mispredict = '0';

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional, branch_type, el, mispredict,
                                '01', Zeros(64), target_address);

        BRBFCCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```





```

// BRBEEException()
// =====
// Called to write exception branch record when BRB is active.

BRBEEException(ExceptionRecord erec, boolean source_valid,
               bits(64) source_address_in,
               bits(64) target_address_in, bits(2) target_el,
               boolean trappedsyscallinst)
bits(64) target_address = target_address_in;
constant Exception except = erec.exceptype;
constant bits(25) iss = erec.syndrome.iss;
case target_el of
  when EL3
    if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then
      return;
  when EL2 if BRBCR_EL2.EXCEPTION == '0' then return;
  when EL1 if BRBCR_EL1.EXCEPTION == '0' then return;

constant boolean target_valid = BranchRecordAllowed(target_el);

if source_valid || target_valid then
  bits(6) branch_type;
  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 CP15RRTTTrap       branch_type = '100011'; // Trap
    when Exception CP14RTTTrap        branch_type = '100011'; // Trap
    when Exception CP14DTTTrap        branch_type = '100011'; // Trap
    when Exception AdvSIMDFFPAccessTrap branch_type = '100011'; // Trap
    when Exception FPIDTrap           branch_type = '100011'; // Trap
    when Exception PACTrap            branch_type = '100011'; // Trap
    when Exception TSTARTAccessTrap   branch_type = '100011'; // Trap
    when Exception CP14RRTTTrap       branch_type = '100011'; // Trap
    when Exception LDST64BTrap        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
    when Exception HypervisorCall     branch_type = '100010'; // Call
    when Exception MonitorCall
      if !trappedsyscallinst then
        branch_type = '100010'; // Call
      else
        branch_type = '100011'; // Trap
    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();

```

```

    when Exception\_Profiling
    when Exception\_GPC
        if iss<20> == '1' then
            else
        otherwise

    branch_type = '100110'; // Inst debug
    branch_type = '101011'; // Inst fault
    branch_type = '101100'; // Data fault
    Unreachable();

bit ccu;
bits(14) cc;
(ccu, cc) = BranchEncCycleCount();
constant bit lastfailed = (if IsFeatureImplemented(FEAT_TME) then BRBFCE_EL1.LASTFAILED
                           else '0');
constant bit transactional = (if source_valid && IsFeatureImplemented(FEAT_TME) &&
                              TSTATE.depth > 0 then '1' else '0');
constant bits(2) el = if target_valid then target_el else '00';
constant bit mispredict = '0';
constant bit sv = if source_valid then '1' else '0';
constant bit tv = if target_valid then '1' else '0';
constant bits(64) source_address = 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);

UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                           branch_type, el, mispredict,
                           sv:tv, source_address, target_address);

BRBFCE_EL1.LASTFAILED = '0';

PMUEvent(PMU_EVENT_BRB_FILTRATE);

return;

```

## Library pseudocode for aarch64/debug/brbe/BRBEEExceptionReturn

```
// BRBEEExceptionReturn()
// =====
// Called to write exception return branch record when BRB is active.

BRBEEExceptionReturn(bits(64) target_address_in, bits(2) source_el,
                     boolean source_valid, bits(64) source_address_in)
    bits(64) target_address = target_address_in;
    case source_el of
        when EL3
            if !IsFeatureImplemented(FEAT_BRBEv1p1) || (MDCR_EL3.E3BREC == MDCR_EL3.E3BREW) then
                return;
        when EL2 if BRBCR_EL2.ERTN == '0' then return;
        when EL1 if BRBCR_EL1.ERTN == '0' then return;

    constant boolean target_valid = BranchRecordAllowed(PSTATE.EL);

    if source_valid || target_valid then
        constant bits(6) branch_type = '000111';
        bit ccu;
        bits(14) cc;
        (ccu, cc) = BranchEncCycleCount();
        constant bit lastfailed = (if IsFeatureImplemented(FEAT_TME) then BRBFCCR_EL1.LASTFAILED
                                   else '0');
        constant bit transactional = (if source_valid && IsFeatureImplemented(FEAT_TME) &&
                                       TSTATE.depth > 0 then '1' else '0');
        constant bits(2) el = if target_valid then PSTATE.EL else '00';
        constant bit mispredict = (if source_valid && BRBEMispredictAllowed() &&
                                    BranchMispredict() then '1' else '0');
        constant bit sv = if source_valid then '1' else '0';
        constant bit tv = if target_valid then '1' else '0';
        constant bits(64) source_address = if source_valid then source_address_in else Zeros(64);
        if !target_valid then
            target_address = Zeros(64);

        UpdateBranchRecordBuffer(ccu, cc, lastfailed, transactional,
                                branch_type, el, mispredict,
                                sv:tv, source_address, target_address);

        BRBFCCR_EL1.LASTFAILED = '0';

        PMUEvent(PMU_EVENT_BRB_FILTRATE);

    return;
```

## Library pseudocode for aarch64/debug/brbe/BRBEFreeze

```
// BRBEFreeze()
// =====
// Generates BRBE freeze event.

BRBEFreeze()
    BRBFCCR_EL1.PAUSED = '1';
    BRBTS_EL1 = GetTimestamp(BRBETimeStamp());
```

## Library pseudocode for aarch64/debug/brbe/BRBEISB

```
// BRBEISB()
// =====
// Handles ISB instruction for BRBE.

BRBEISB()
    constant boolean branch_conditional = FALSE;
    BRBEBranch(BranchType\_DIR, branch_conditional, PC64 + 4);
```

## Library pseudocode for aarch64/debug/brbe/BRBEMispredictAllowed

```
// BRBEMispredictAllowed()
// =====
// Returns TRUE if the recording of branch misprediction is allowed,
// FALSE otherwise.

boolean BRBEMispredictAllowed()
    if HaveEL\(EL2\) && BRBCR_EL2.MPRED == '0' then return FALSE;
    if BRBCR_EL1.MPRED == '0' then return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/debug/brbe/BRBETimeStamp

```
// BRBETimeStamp()
// =====
// Returns captured timestamp.

TimeStamp BRBETimeStamp()
    if HaveEL\(EL2\) then
        TS_el2 = BRBCR_EL2.TS;
        if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
            // Reserved value
            (-, TS_el2) = ConstrainUnpredictableBits\(Unpredictable\_EL2TIMESTAMP, 2\);
        case TS_el2 of
            when '00'
                // Falls out to check BRBCR_EL1.TS
            when '01'
                return TimeStamp\_Virtual;
            when '10'
                assert IsFeatureImplemented(FEAT_ECV); // Otherwise ConstrainUnpredictableBits
                                                         // removes this case
                return TimeStamp\_OffsetPhysical;
            when '11'
                return TimeStamp\_Physical;

    TS_el1 = BRBCR_EL1.TS;
    if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits\(Unpredictable\_EL1TIMESTAMP, 2\);
    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
```

## Library pseudocode for aarch64/debug/brbe/BRB\_IALL

```
// BRB_IALL()
// =====
// Called to perform invalidation of branch records

BRB_IALL()
    for i = 0 to GetBRBENumRecords\(\) - 1
        Records_SRC[i] = Zeros\(64\);
        Records_TGT[i] = Zeros\(64\);
        Records_INF[i] = Zeros\(64\);
```

## Library pseudocode for aarch64/debug/brbe/BRB\_INJ

```
// BRB_INJ()
// =====
// Called to perform manual injection of branch records.

BRB_INJ()
    UpdateBranchRecordBuffer(BRBINFINJ_EL1.CCU, BRBINFINJ_EL1.CC, BRBINFINJ_EL1.LASTFAILED,
                             BRBINFINJ_EL1.T, BRBINFINJ_EL1.TYPE, BRBINFINJ_EL1.EL,
                             BRBINFINJ_EL1.MPRED, BRBINFINJ_EL1.VALID, BRBSRCINJ_EL1.ADDRESS,
                             BRBTGTINJ_EL1.ADDRESS);

    BRBINFINJ_EL1 = bits(64) UNKNOWN;
    BRBSRCINJ_EL1 = bits(64) UNKNOWN;
    BRBTGTINJ_EL1 = bits(64) UNKNOWN;

    if ConstrainUnpredictableBool(Unpredictable\_BRBFILTRATE) then
        PMUEvent(PMU_EVENT_BRB_FILTRATE);
```

## 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 result 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.

(bit, bits(14)) BranchEncCycleCount()
    cc = BranchRawCycleCount();
    if !BRBCycleCountingEnabled() || FirstBranchAfterProhibited() then
        return ('1', Zeros(14));

    // 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 = IsZero(E) ? UInt(M) : 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.

    bits(6) E;
    bits(8) M;

    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 do
            E = E + 1;
            cc = cc DIV 2;
        M = cc<7:0>;

    return ('0', E:M);
```

### Library pseudocode for aarch64/debug/brbe/BranchMispredict

```
// BranchMispredict()
// =====
// Returns TRUE if the branch being executed was mispredicted, FALSE otherwise.

boolean BranchMispredict();
```

### 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.

integer BranchRawCycleCount();
```

### Library pseudocode for aarch64/debug/brbe/BranchRecordAllowed

```
// BranchRecordAllowed()
// =====
// Returns TRUE if branch recording is allowed, FALSE otherwise.

boolean BranchRecordAllowed(bits(2) el)
    if ELUsingAArch32(el) then
        return FALSE;

    if BRBFCR_EL1.PAUSED == '1' then
        return FALSE;

    if el == EL3 && IsFeatureImplemented(FEAT_BRBEv1p1) then
        return (MDCR_EL3.E3BREC != MDCR_EL3.E3BREW);

    if HaveEL(EL3) && (MDCR_EL3.SBRBE == '00' ||
        (CurrentSecurityState() == SS\_Secure && MDCR_EL3.SBRBE == '01')) then
        return FALSE;

    case el of
        when EL3 return FALSE; // FEAT_BRBEv1p1 not implemented
        when EL2 return BRBCR_EL2.E2BRE == '1';
        when EL1 return BRBCR_EL1.E1BRE == '1';
        when EL0
            if EL2Enabled() && HCR_EL2.TGE == '1' then
                return BRBCR_EL2.E0HBRE == '1';
            else
                return BRBCR_EL1.E0BRE == '1';
```

### Library pseudocode for aarch64/debug/brbe/Contents

```
// Contents of the Branch Record Buffer
//=====

array [0..63] of BRBSRC_EL1_Type Records_SRC;

array [0..63] of BRBTGT_EL1_Type Records_TGT;

array [0..63] of BRBINF_EL1_Type Records_INF;
```

### Library pseudocode for aarch64/debug/brbe/FilterBranchRecord

```
// FilterBranchRecord()
// =====
// Returns TRUE if the branch record is not filtered out, FALSE otherwise.

boolean FilterBranchRecord(BranchType br, boolean cond)
    case br of
        when BranchType\_DIRCALL
            return BRBFCE_EL1.DIRCALL != BRBFCE_EL1.EnI;
        when BranchType\_INDCALL
            return BRBFCE_EL1.INDCALL != BRBFCE_EL1.EnI;
        when BranchType\_RET
            return BRBFCE_EL1.RTN != BRBFCE_EL1.EnI;
        when BranchType\_DIR
            if cond then
                return BRBFCE_EL1.CONDDIR != BRBFCE_EL1.EnI;
            else
                return BRBFCE_EL1.DIRECT != BRBFCE_EL1.EnI;
        when BranchType\_INDIR
            return BRBFCE_EL1.INDIRECT != BRBFCE_EL1.EnI;
        otherwise
            Unreachable();
    return FALSE;
```

### Library pseudocode for aarch64/debug/brbe/FirstBranchAfterProhibited

```
// FirstBranchAfterProhibited()
// =====
// Returns TRUE if branch recorded is the first branch after a prohibited region,
// FALSE otherwise.

boolean FirstBranchAfterProhibited();
```

### Library pseudocode for aarch64/debug/brbe/GetBRBNumRecords

```
// GetBRBNumRecords()
// =====
// Returns the number of branch records implemented.

integer GetBRBNumRecords()
    assert UInt(BRBIDR0_EL1.NUMREC) IN {0x08, 0x10, 0x20, 0x40};
    return integer IMPLEMENTATION_DEFINED "Number of BRB records";
```

## Library pseudocode for aarch64/debug/brbe/Getter

```
// Getter functions for branch records
// =====
// Functions used by MRS instructions that access branch records

BRBSRC_EL1_Type BRBSRC_EL1[integer n]
  assert n IN {0..31};
  constant integer record_index = UInt(BRBFCCR_EL1.BANK:n<4:0>);
  if record_index < GetBRBENumRecords() then
    return Records_SRC[record_index];
  else
    return Zeros(64);

BRBTGT_EL1_Type BRBTGT_EL1[integer n]
  assert n IN {0..31};
  constant integer record_index = UInt(BRBFCCR_EL1.BANK:n<4:0>);
  if record_index < GetBRBENumRecords() then
    return Records_TGT[record_index];
  else
    return Zeros(64);

BRBINF_EL1_Type BRBINF_EL1[integer n]
  assert n IN {0..31};
  constant integer record_index = UInt(BRBFCCR_EL1.BANK:n<4:0>);
  if record_index < GetBRBENumRecords() then
    return Records_INF[record_index];
  else
    return Zeros(64);
```

## Library pseudocode for aarch64/debug/brbe/ShouldBRBEFreeze

```
// ShouldBRBEFreeze()
// =====
// Returns TRUE if the BRBE freeze event conditions have been met, and FALSE otherwise.

boolean ShouldBRBEFreeze()
  if !BranchRecordAllowed(PSTATE.EL) then
    return FALSE;

  boolean include_r1;
  boolean include_r2;
  constant boolean include_r3 = FALSE;

  if HaveEL(EL2) then
    include_r1 = (BRBCR_EL1.FZP == '1');
    include_r2 = (BRBCR_EL2.FZP == '1');
  else
    include_r1 = TRUE;
    include_r2 = TRUE;

  return CheckPMUOverflowCondition(PMUOverflowCondition_BRBEFreeze,
    include_r1, include_r2, include_r3);
```



## Library pseudocode for aarch64/debug/brbe/UpdateBranchRecordBuffer

```
// UpdateBranchRecordBuffer()
// =====
// Add a new Branch record to the buffer.

UpdateBranchRecordBuffer(bit ccu, bits(14) cc, bit lastfailed, bit transactional,
                        bits(6) branch_type, bits(2) el, bit mispredict, bits(2) valid,
                        bits(64) source_address, bits(64) target_address)
// Shift the Branch Records in the buffer
for i = GetBRBENumRecords() - 1 downto 1
    Records_SRC[i] = Records_SRC[i - 1];
    Records_TGT[i] = Records_TGT[i - 1];
    Records_INF[i] = Records_INF[i - 1];

Records_INF[0].CCU      = ccu;
Records_INF[0].CC       = cc;

Records_INF[0].EL       = el;
Records_INF[0].VALID    = valid;
Records_INF[0].T        = transactional;
Records_INF[0].LASTFAILED = lastfailed;
Records_INF[0].MPRED    = mispredict;
Records_INF[0].TYPE     = branch_type;

Records_SRC[0] = source_address;
Records_TGT[0] = target_address;

return;
```

## 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.

BreakpointInfo AArch64.BreakpointMatch(integer n, bits(64) vaddress, AccessDescriptor accdesc,
                                         integer size)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumBreakpointsImplemented();
    BreakpointInfo brkptinfo;

    linking_enabled = (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;

    enabled      = IsBreakpointEnabled(n);
    linked       = DBGBCR_EL1[n].BT == '0x01';
    isbreakpnt  = TRUE;
    linked_to    = FALSE;
    from_linking_enabled = FALSE;
    lbnx        = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1[n].LBNX else '00';
    linked_n     = UInt(lbnx : DBGBCR_EL1[n].LBN);
    ssce         = if IsFeatureImplemented(FEAT_RME) then DBGBCR_EL1[n].SSCE else '0';
    state_match  = AArch64.StateMatch(DBGBCR_EL1[n].SSC, ssce, DBGBCR_EL1[n].HMC,
                                     DBGBCR_EL1[n].PMC, linked, linked_n, isbreakpnt,
                                     vaddress, accdesc);

    (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.
        (-, match_i) = AArch64.BreakpointValueMatch(n, vaddress + 2, linked_to, isbreakpnt,
                                                    from_linking_enabled);

        if !value_match && match_i then
            value_match = ConstrainUnpredictableBool(Unpredictable\_BPMATCHHALF);

    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);

    if !(state_match && enabled) then
        brkptinfo.bptype = BreakpointType\_Inactive;
        brkptinfo.match = FALSE;
    else
        brkptinfo.bptype = bp_type;
        brkptinfo.match = value_match;
    return brkptinfo;
```



```

// 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.

(BreakpointType, boolean) AArch64.BreakpointValueMatch(integer n_in, bits(64) vaddress,
                                                         boolean linked_to, boolean isbreakpnt,
                                                         boolean from_linking_enabled)

// "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.
integer n = n_in;
Constraint c;

// 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);

// 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);

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR_EL1[n].BT;
bt2 = if IsFeatureImplemented(FEAT_ABLE) then DBGBCR_EL1[n].BT2 else '0';

(c, bt2, dbgtype) = AArch64.ReservedBreakpointType(n, bt2, dbgtype);
if c == Constraint DISABLED then return (BreakpointType Inactive, FALSE);
// Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr      = (dbgtype == '0x0x');
mismatch        = (dbgtype == '010x');
match_vmid       = (dbgtype == '10xx');
match_cid        = (dbgtype == '001x');
match_cid1       = (dbgtype IN {'101x', 'x11x'});
match_cid2       = (dbgtype == '11xx');
linking_enabled  = (dbgtype IN {'xx11', '1xx1'} || bt2 == '1');

// 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);

// If called from BreakpointMatch return FALSE for breakpoint with linking enabled.
if !linked_to && linking_enabled then
    return (BreakpointType Inactive, FALSE);

constant boolean linked = (dbgtype == '0x01');
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);

// 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);

// A breakpoint programmed for address mismatch does not match in AArch32 state.
if mismatch && UsingAArch32() then
    return (BreakpointType_Inactive, FALSE);

boolean bvr_match = FALSE;
boolean bxvr_match = FALSE;
BreakpointType bp_type;
integer mask;

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
    (c, 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.

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);
else
    // A stand-alone mismatch of a single address is not supported.
    if mismatch then
        return (BreakpointType_Inactive, FALSE);

else
    mask = 0;

// Do the comparison.
if match_addr then
    boolean byte_select_match;
    constant integer byte = 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

// 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.
constant AddressSize dbgtop = DebugAddrTop();
constant boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGBCR_EL1[n]<63:dbgtop>) &&
    !IsZero(DBGBCR_EL1[n]<63:dbgtop>) &&
    ConstrainUnpredictableBool(Unpredictable DBGxVR_RESS));
constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
constant integer cmpls = 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 DBG_BVR_EL1[n] are not zero, the behavior
        // is CONSTRAINED UNPREDICTABLE.
        constant integer masktop = mask - 1;
        if bvr_match && !IsZero(DBG_BVR_EL1[n]<masktop:2>) then
            bvr_match = ConstrainUnpredictableBool(Unpredictable\_BPMASKEDBITS);

elseif match_cid then
    if IsInHost() then
        bvr_match = (CONTEXTIDR_EL2<31:0> == DBG_BVR_EL1[n]<31:0>);
    else
        bvr_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBG_BVR_EL1[n]<31:0>);

elseif match_cid1 then
    bvr_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() &&
        CONTEXTIDR_EL1<31:0> == DBG_BVR_EL1[n]<31:0>);

if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;

    if !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBG_BVR_EL1[n]<39:32>, 16);
    else
        vmid = VTBR_EL2.VMID;
        bvr_vmid = DBG_BVR_EL1[n]<47:32>;

    bxvr_match = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() && vmid == bvr_vmid);

elseif match_cid2 then
    bxvr_match = (PSTATE.EL != EL3 && EL2Enabled() &&
        DBG_BVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

value_match = (!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);
    lbnx = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGBCR_EL1[n].LBNX else '00';
    linked_linked_n = UInt(lbnx : DBGBCR_EL1[n].LBN);
    boolean linked_value_match;
    linked_vaddress = bits(64) UNKNOWN;
    linked_linked_to = TRUE;
    linked_isbreakpnt = TRUE;
    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;

if match_addr && !mismatch then
    bp_type = BreakpointType\_AddrMatch;
elseif match_addr && mismatch then
    bp_type = BreakpointType\_AddrMismatch;
elseif match_vmid || match_cid || match_cid1 || match_cid2 then
    bp_type = BreakpointType\_CtxtMatch;
else
    Unreachable();

return (bp_type, value_match);

```

## Library pseudocode for aarch64/debug/breakpoint/AArch64.ReservedBreakpointType

```
// AArch64.ReservedBreakpointType()
// =====
// Checks if the given DBGBCR<n>_EL1.{BT2,BT} values are reserved and will
// generate Constrained Unpredictable behavior, otherwise returns Constraint_NONE.

(Constraint, bit, bits(4)) AArch64.ReservedBreakpointType(integer n, bit bt2_in ,bits(4) bt_in)
    bit bt2          = bt2_in;
    bits(4) bt        = bt_in;
    boolean reserved = FALSE;
    context_aware = IsContextAwareBreakpoint(n);

    if bt2 == '0' then
        // Context matching
        if bt != '0x0x' && !context_aware then
            reserved = TRUE;

        // EL2 extension
        if bt == '1xxx' && !HaveEL(EL2) then
            reserved = TRUE;

        // Context matching
        if (bt IN {'011x','11xx'}) && !IsFeatureImplemented(FEAT_VHE) &&
            !IsFeatureImplemented(FEAT_Debugv8p2)) then
            reserved = TRUE;

        // Reserved
        if bt == '010x' && !IsFeatureImplemented(FEAT_BWE) && !HaveAArch32EL(EL1) then
            reserved = TRUE;
    else
        // Reserved
        if bt != '0x0x' then
            reserved = TRUE;

    if reserved then
        Constraint c;
        (c, <bt2, bt>) = ConstrainUnpredictableBits(Unpredictable\_RESBPTYPE, 5);
        assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
        if c == Constraint\_DISABLED then
            return (c, bit UNKNOWN, bits(4) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint\_NONE, bt2, bt);
```





```

// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) ssc_in, bit ssce_in, bit hmc_in,
                           bits(2) pxc_in, boolean linked_in, integer linked_n_in,
                           boolean isbreakpnt, bits(64) vaddress, AccessDescriptor accdesc)
if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';

// "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.
bits(2) ssc      = ssc_in;
bit ssce         = ssce_in;
bit hmc          = hmc_in;
bits(2) pxc      = pxc_in;
boolean linked   = linked_in;
integer linked_n = linked_n_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, ssc, ssce, hmc, pxc) = CheckValidStateMatch(ssc, ssce, hmc, pxc, isbreakpnt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the hmc, ssc, ssce, pxc values are either valid or the values returned by
// CheckValidStateMatch are valid.

EL3_match = HaveEL(EL3) && hmc == '1' && ssc<0> == '0';
EL2_match = HaveEL(EL2) && ((hmc == '1' && (ssc:pxc != '1000')) || ssc == '11');
EL1_match = pxc<0> == '1';
EL0_match = pxc<1> == '1';

boolean priv_match;
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;

// Security state match
boolean ss_match;
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;

boolean linked_match = 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
    (first_ctx_cmp, last_ctx_cmp) = 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

```

```

if linked then
    linked_to = TRUE;
    BreakpointType bp_type;
    from_linking_enabled = 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;

return priv_match && ss_match && (!linked || linked_match);

```

### Library pseudocode for aarch64/debug/breakpoint/DebugAddrTop

```

// DebugAddrTop()
// =====
// Returns the value for the top bit used in Breakpoint and Watchpoint address comparisons.

AddressSize DebugAddrTop()
    if IsFeatureImplemented(FEAT_LVA3) then
        return 55;
    elsif IsFeatureImplemented(FEAT_LVA) then
        return 52;
    else
        return 48;

```

### Library pseudocode for aarch64/debug/breakpoint/EffectiveMDSELR\_EL1\_BANK

```

// EffectiveMDSELR_EL1_BANK()
// =====
// Return the effective value of MDSELR_EL1.BANK.

bits(2) EffectiveMDSELR_EL1_BANK()
    // If 16 or fewer breakpoints and 16 or fewer watchpoints are implemented,
    // then the field is RES0.
    constant integer num_bp = NumBreakpointsImplemented();
    constant integer num_wp = NumWatchpointsImplemented();
    if num_bp <= 16 && num_wp <= 16 then
        return '00';

    // 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';

    bits(2) bank = 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(Unpredictable_RESMDSELR, 2);
        // The value returned by ConstrainUnpredictableBits must be a not-reserved value

return bank;

```

## Library pseudocode for aarch64/debug/breakpoint/IsBreakpointEnabled

```
// IsBreakpointEnabled()
// =====
// Returns TRUE if the effective value of DBGBCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsBreakpointEnabled(integer n)
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0')) then
        return FALSE;

    return DBGBCR_EL1[n].E == '1';
```

## 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.

boolean SelfHostedExtendedBPWPEEnabled()
    if NumBreakpointsImplemented() <= 16 && NumWatchpointsImplemented() <= 16 then
        return FALSE;

    if ((HaveEL(EL3) && MDCR_EL3.EBWE == '0') ||
        (EL2Enabled() && MDCR_EL2.EBWE == '0')) then
        return FALSE;

    return MDSCR_EL1.EMBWE == '1';
```

## Library pseudocode for aarch64/debug/ebep/CheckForPMUException

```
// CheckForPMUException()
// =====
// Take a PMU exception if enabled, permitted, and unmasked.

CheckForPMUException()
    boolean enabled;
    bits(2) target_el;
    boolean pmu_exception;
    boolean synchronous;
    (enabled, target_el) = PMUExceptionEnabled();
    if !enabled || PMUExceptionMasked(target_el, PSTATE.EL, PSTATE.PM) then
        pmu_exception = FALSE;
    elsif IsFeatureImplemented(FEAT_SEBEP) && PSTATE.PPEND == '1' then
        pmu_exception = TRUE;
        synchronous = TRUE;
    else
        constant boolean include_r1 = TRUE;
        constant boolean include_r2 = TRUE;
        constant boolean include_r3 = TRUE;
        pmu_exception = CheckPMUOverflowCondition(PMUOverflowCondition PMUException,
                                                include_r1, include_r2, include_r3);
        synchronous = FALSE;
    if pmu_exception then
        constant bits(5) fsc = '00000';
        TakeProfilingException(target_el, fsc, synchronous);
```

## Library pseudocode for aarch64/debug/ebep/ExceptionReturnPPEND

```
// ExceptionReturnPPEND()
// =====
// Sets ShouldSetPPEND to the value to write to PSTATE.PPEND
// on an exception return.
// This function is called before any change in Exception level.

ExceptionReturnPPEND(bits(64) spsr)
    boolean enabled_at_source;
    boolean masked_at_source;
    if spsr<33> == '1' then                                // SPSR.PPEND
        bits(2) target_except;
        (enabled_at_source, target_except) = PMUExceptionEnabled\(\);
        masked_at_source = PMUExceptionMasked(target_except, PSTATE.EL, PSTATE.PM);

        bits(2) target_eret;
        if IllegalExceptionReturn(spsr) then
            target_eret = PSTATE.EL;
        else
            boolean valid;
            (valid, target_eret) = ELFromSPSR(spsr);
            assert valid;

        constant bit target_pm = spsr<32>;                // SPSR.PM
        constant boolean masked_at_dest = PMUExceptionMasked(target_except, target_eret, target_pm);
        if enabled_at_source && masked_at_source && !masked_at_dest then
            PSTATE.PPEND = '1';
            ShouldSetPPEND = FALSE;
            // PSTATE.PPEND will not be changed again by this instruction.

        // If PSTATE.PPEND has not been set by this function, ShouldSetPPEND is
        // unchanged, meaning PSTATE.PPEND might either be set by the current instruction
        // causing a counter overflow, or cleared to zero at the end of instruction.

    return;
```

## Library pseudocode for aarch64/debug/ebep/IsSupportingPMUSynchronousMode

```
// IsSupportingPMUSynchronousMode()
// =====
// Returns TRUE if the event support synchronous mode,
// and FALSE otherwise.

boolean IsSupportingPMUSynchronousMode(bits(16) pmuevent);
```

## 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.

(boolean, bits(2)) PMUExceptionEnabled()

    if !IsFeatureImplemented(FEAT_EBEP) then
        return (FALSE, bits(2) UNKNOWN);

    boolean enabled;
    bits(2) target = bits(2) UNKNOWN;

    if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
        enabled = MDCR_EL3.PMEE == '11';
        if enabled then target = EL3;

    elsif EL2Enabled() && MDCR_EL2.PMEE != '01' then
        enabled = MDCR_EL2.PMEE == '11';
        if enabled then target = EL2;

    else
        bits(2) pmee_el1 = PMECR_EL1.PMEE;
        if pmee_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMEE, 2);
            assert c IN {Constraint DISABLED, Constraint UNKNOWN};
            if c == Constraint DISABLED then pmee_el1 = '10';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

        enabled = pmee_el1 == '11';
        if enabled then
            target = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;

    return (enabled, target);
```

## 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.

boolean PMUExceptionMasked(bits(2) target_el, bits(2) from_el, bit pm)
    assert IsFeatureImplemented(FEAT_EBEP);

    if Halted() || Restarting() then
        return TRUE;
    elsif UInt(target_el) < UInt(from_el) then
        return TRUE;
    elsif from_el == EL2 && target_el == EL2 && MDCR_EL2.PMEE != '11' then
        return TRUE;
    elsif target_el == from_el && (PMECR_EL1.KPME == '0' || pm == '1') then
        return TRUE;

    return FALSE;
```

## Library pseudocode for aarch64/debug/ebep/PMUIInterruptEnabled

```
// PMUIInterruptEnabled()
// =====
// Return TRUE if the PMU interrupt request (PMUIRQ) is enabled, FALSE otherwise.

boolean PMUIInterruptEnabled()
    if !IsFeatureImplemented(FEAT_EBEP) then
        return TRUE;

    boolean enabled;

    if HaveEL(EL3) && MDCR_EL3.PMEE != '01' then
        enabled = MDCR_EL3.PMEE == '00';

    elsif EL2Enabled() && MDCR_EL2.PMEE != '01' then
        enabled = MDCR_EL2.PMEE == '00';

    else
        bits(2) pmee_el1 = PMECR_EL1.PMEE;
        if pmee_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmee_el1) = ConstrainUnpredictableBits(Unpredictable\_RESPMEE, 2);
            assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
            if c == Constraint\_DISABLED then pmee_el1 = '10';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
            enabled = pmee_el1 == '00';

    return enabled;
```

## Library pseudocode for aarch64/debug/ebep/inst\_addr\_executed

```
bits(64) inst_addr_executed;
```

## Library pseudocode for aarch64/debug/ebep/sync\_counter\_overflowed

```
boolean sync_counter_overflowed;
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptions

```
// AArch64.GenerateDebugExceptions()
// =====

boolean AArch64.GenerateDebugExceptions()
    ss = CurrentSecurityState();
    return AArch64.GenerateDebugExceptionsFrom(PSTATE.EL, ss, PSTATE.D);
```

## Library pseudocode for aarch64/debug/enables/AArch64.GenerateDebugExceptionsFrom

```
// AArch64.GenerateDebugExceptionsFrom()
// =====

boolean AArch64.GenerateDebugExceptionsFrom(bits(2) from_el, SecurityState from_state, bit mask)

    if OSLSR_EL1.OSLK == '1' || DoubleLockStatus() || Halted() then
        return FALSE;

    route_to_el2 = (HaveEL(EL2) && (from_state != SS\_Secure || IsSecureEL2Enabled()) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));
    target = (if route_to_el2 then EL2 else EL1);
    boolean enabled;
    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';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDSCR_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt(target) > UInt(from_el);

    return enabled;
```

## 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

AArch64.TRCIT(bits(64) Xt)
    ss = CurrentSecurityState();
    if TraceInstrumentationAllowed(ss, PSTATE.EL) then
        TraceInstrumentation(Xt);
```

## Library pseudocode for aarch64/debug/ite/TraceInstrumentation

```
// TraceInstrumentation()
// =====
// Generates an instrumentation trace packet
// containing the value of the register passed as an argument

TraceInstrumentation(bits(64) Xt);
```

## Library pseudocode for aarch64/debug/pmu/AArch64.IncrementCycleCounter

```
// AArch64.IncrementCycleCounter()
// =====
// Increment the cycle counter and possibly set overflow bits.

AArch64.IncrementCycleCounter()
    if !CountPMUEvents(CYCLE_COUNTER_ID) then return;
    bit d = PMCR_EL0.D;    // Check divide-by-64
    bit lc = PMCR_EL0.LC;
    boolean lc_enabled;
    (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';
    if d == '1' && !HasElapsed64Cycles() then return;

    constant integer old_value = UInt(PMCCNTR_EL0);
    constant integer new_value = old_value + 1;
    PMCCNTR_EL0 = new_value<63:0>;

    constant integer ovflw = if HaveAArch32() && lc == '0' then 32 else 64;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0.C = '1';
        PMOVSLR_EL0.C = '1';

    return;
```



## 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.

integer AArch64.IncrementEventCounter(integer idx, integer increment_in, integer Vm)
    integer old_value;
    integer new_value;

    old_value = UInt(PMEVCNTR_EL0[idx]);
    constant integer increment = PMUCountValue(idx, increment_in, Vm);
    new_value = old_value + increment;

    bit lp;
    if IsFeatureImplemented(FEAT_PMUv3p5) then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        boolean pmuexception_enabled;
        (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
        else
            lp = '0';
            PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>, 64);
            constant integer ovflw = if lp == '1' then 64 else 32;

            if old_value<64:ovflw> != new_value<64:ovflw> then
                PMOVSSET_EL0<idx> = '1';
                PMOVSLR_EL0<idx> = '1';
                // Check for the CHAIN event from an even counter
                if (idx<0> == '0' && idx + 1 < NUM_PMU_COUNTERS &&
                    (!IsFeatureImplemented(FEAT_PMUv3p5) || lp == '0')) then
                    PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
            if (IsFeatureImplemented(FEAT_SEBEP) &&
                IsSupportingPMUSynchronousMode(PMEVTYPEPER_EL0[idx].evtCount) &&
                PMINTENSET_EL1<idx> == '1' && PMOVSLR_EL0<idx> == '1' && increment != 0) then
                    SyncCounterOverflowed = TRUE;

            return increment;
```

## 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.

AArch64.PMUCycle()
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    constant integer counters = NUM_PMU_COUNTERS;
    integer Vm = 0;
    if counters != 0 then
        for idx = 0 to counters - 1
            if CountPMUEvents(idx) then
                constant integer accumulated = PMUEventAccumulator[idx];
                if (idx MOD 2) == 0 then Vm = 0;
                Vm = AArch64.IncrementEventCounter(idx, accumulated, Vm);
                PMUEventAccumulator[idx] = 0;
            AArch64.IncrementCycleCounter();
        CheckForPMUOverflow();
```

## Library pseudocode for aarch64/debug/profilingexception/TakeProfilingException

```
// TakeProfilingException()
// =====
// Takes a Profiling exception.

TakeProfilingException(bits(2) target_el, bits(5) fsc, boolean synchronous)
    ExceptionRecord except = ExceptionSyndrome(Exception Profiling);
    except.syndrome.iss<5:1> = fsc;
    if synchronous then
        except.syndrome.iss<0> = '1';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CheckForSPEException

```
// CheckForSPEException()
// =====
// Take an SPE Profiling exception if pending, permitted, and unmasked.

CheckForSPEException()
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return;

    if Halted\(\) || Restarting\(\) then
        return;

    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;
    boolean route_to_el1 = FALSE;

    if HaveEL\(EL3\) && MDCR_EL3.PMSEE == '1x' then
        constant boolean pending = PMBSR_EL3.S == '1';
        constant boolean masked = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner\(\);
    constant boolean in_owning_ss = IsCurrentSecurityState(owning_ss);

    if EffectivePMSCR\_EL2\_EE\(\) == '1x' then
        constant boolean pending = PMBSR_EL2.S == '1';
        constant boolean masked = (!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;

    if EffectivePMSCR\_EL1\_EE\(\) == '11' then
        constant boolean pending = PMBSR_EL1.S == '1';
        constant boolean masked = (!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;

    constant bits(5) fsc = '00001'; // SPE exception
    constant boolean synchronous = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR1

```
// CollectContextIDR1()
// =====

boolean CollectContextIDR1()
    if !StatisticalProfilingEnabled\(\) then return FALSE;
    if PSTATE.EL == EL2 then return FALSE;
    if EL2Enabled\(\) && HCR_EL2.TGE == '1' then return FALSE;
    return PMSCR_EL1.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectContextIDR2

```
// CollectContextIDR2()
// =====

boolean CollectContextIDR2()
    if !StatisticalProfilingEnabled() then return FALSE;
    if !EL2Enabled() then return FALSE;
    return PMSCR_EL2.CX == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectPhysicalAddress

```
// CollectPhysicalAddress()
// =====

boolean CollectPhysicalAddress()
    if !StatisticalProfilingEnabled() then return FALSE;
    (owning_ss, owning_el) = 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';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/CollectTimeStamp

```
// CollectTimeStamp()
// =====

TimeStamp CollectTimeStamp()
    if !StatisticalProfilingEnabled\(\) then return TimeStamp\_None;
    (-, owning_el) = ProfilingBufferOwner();

    if owning_el == EL2 then
        if PMSCR_EL2.TS == '0' then return TimeStamp\_None;
    else
        if PMSCR_EL1.TS == '0' then return TimeStamp\_None;

    bits(2) PCT_el1;
    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(Unpredictable\_PMSCR\_PCT, 2);
    if EL2Enabled() then
        bits(2) PCT_el2;
        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(Unpredictable\_PMSCR\_PCT, 2);
    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;
        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;
        otherwise
            Unreachable();

    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();
```

## 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.

bits(2) DefaultSPEEvent()
    return ReportSPEEvent(Zeros(6), bits(6) UNKNOWN);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/EffectivePMBLIMITR\_EL1\_nVM

```
// EffectivePMBLIMITR_EL1_nVM()
// =====

bit EffectivePMBLIMITR_EL1_nVM()
    if !IsFeatureImplemented(FEAT_SPE_nVM) then
        return '0';
    if HaveEL(EL2) then
        (owning_ss, owning_el) = ProfilingBufferOwner();
        if ((owning_ss != SS_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
            PMSCR_EL2.EnVM == '0') then
            return '0';
    return PMBLIMITR_EL1.nVM;
```

## 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.

bits(2) EffectivePMSCR_EL1_EE()
    if EffectivePMSCR_EL2_EE() == '00' then
        return '00';

    bits(2) ee = PMSCR_EL1.EE;
    if ee IN {'01', '10'} then // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee<0> = ee<1>;
        else
            Constraint c;
            (c, ee) = ConstrainUnpredictableBits(Unpredictable_RESPMSEE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then
                ee = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

    return ee;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/EffectivePMSCR\_EL2\_EE

```
// EffectivePMSCR_EL2_EE()
// =====
// Return the Effective value of PMSCR_EL2.EE.

bits(2) EffectivePMSCR_EL2_EE()
    if !IsFeatureImplemented(FEAT_SPE_EXC) then
        return '00';

    if HaveEL(EL3) && MDCR_EL3.PMSEE == '00' then
        return '00';

    constant boolean check_el2 = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
                                                IsSecureEL2Enabled());
    return if check_el2 then PMSCR_EL2.EE else '01';
```

### Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR\_EL1\_FSC

```
// GetPMBSR_EL1_FSC()
// =====
// Query the PMBSR_EL1.FSC field.

bits(6) GetPMBSR_EL1_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL1<5:0>;
    return FSC;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR\_EL2\_FSC

```
// GetPMBSR_EL2_FSC()
// =====
// Query the PMBSR_EL2.FSC field.

bits(6) GetPMBSR_EL2_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL2<5:0>;
    return FSC;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/GetPMBSR\_EL3\_FSC

```
// GetPMBSR_EL3_FSC()
// =====
// Query the PMBSR_EL3.FSC field.

bits(6) GetPMBSR_EL3_FSC()
    bits(6) FSC;

    FSC = PMBSR_EL3<5:0>;
    return FSC;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/OtherSPEManagementEvent

```
// OtherSPEManagementEvent()
// =====
// Report an Other buffer management event, with the status code 'bsc'

OtherSPEManagementEvent(bits(6) bsc)
    constant bits(2) target_el = 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(bsc, 16);
        PMBSR\_EL[target_el].MSS2 = Zeros(24);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/PMBSR\_EL

```
// PMBSR_EL[] - assignment form
// =====

PMBSRType PMBSR_EL[bits(2) el]
  bits(64) r;
  case el of
    when EL1    r = PMBSR_EL1;
    when EL2    r = PMBSR_EL2;
    when EL3    r = PMBSR_EL3;
    otherwise  Unreachable();
  return r;

// PMBSR_EL[] - non-assignment form
// =====

PMBSR_EL[bits(2) el] = bits(64) value
  constant bits(64) r = value;
  case el of
    when EL1    PMBSR_EL1 = r;
    when EL2    PMBSR_EL2 = r;
    when EL3    PMBSR_EL3 = r;
    otherwise  Unreachable();
  return;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
  if !IsFeatureImplemented(FEAT_SPE) || Halted\(\) then
    return FALSE;

  (owning_ss, owning_el) = ProfilingBufferOwner\(\);
  constant bits(2) state_bits = EffectiveSCR\_EL3\_NSE\(\) : EffectiveSCR\_EL3\_NS\(\);

  boolean state_match;
  case owning_ss of
    when SS\_Secure    state_match = state_bits == '00';
    when SS\_NonSecure state_match = state_bits == '01';
    when SS\_Realm     state_match = state_bits == '11';

  constant boolean stopped = SPEProfilingStopped\(\);

  return (!ELUsingAArch32(owning_el) && state_match &&
    PMBLIMITR_EL1.E == '1' && !stopped);
```



## Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(SecurityState, bits(2)) ProfilingBufferOwner()
    SecurityState owning_ss;

    if HaveEL(EL3) then
        bits(3) state_bits;
        if IsFeatureImplemented(FEAT_RME) then
            state_bits = MDCR_EL3.<NSPBE,NSPB>;
            if (state_bits == '10x' ||
                (!IsFeatureImplemented(FEAT_SEL2) && state_bits == '00x')) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits(Unpredictable\_RESERVEDNSxB, 3);
        else
            state_bits = '0' : MDCR_EL3.NSPB;

        case state_bits of
            when '00x' owning_ss = SS\_Secure;
            when '01x' owning_ss = SS\_NonSecure;
            when '11x' owning_ss = SS\_Realm;
        else
            owning_ss = if SecureOnlyImplementation() then SS\_Secure else SS\_NonSecure;

        bits(2) owning_el;
        if HaveEL(EL2) && (owning_ss != SS\_Secure || IsSecureEL2Enabled()) then
            owning_el = if MDCR_EL2.E2PB == '00' then EL2 else EL1;
        else
            owning_el = EL1;

        return (owning_ss, owning_el);
```

## 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.

ProfilingSynchronizationBarrier();
```

## 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().

bits(2) ReportSPEEvent(bits(6) ec_bits, bits(6) fsc_bits)
    bits(2) target_el;
    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;

    if IsFeatureImplemented(FEAT_SPE_EXC) then
        constant boolean s1fault = (ec_bits == '100100'); // Stage 1 fault
        constant boolean s2fault = (ec_bits == '100101'); // Stage 2 fault

        boolean gpcfault, gpfault;
        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;

        SecurityState owning_ss;
        bits(2) owning_el;
        (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'));

        if EffectivePMSCR\_EL2\_EE() == '1x' then
            route_to_el2 = (PMSCR_EL2.EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2.GPF == '1'));

        if route_to_el3 then
            target_el = EL3;
        elsif route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    return target_el;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddByteToRecord

```
// SPEAddByteToRecord()
// =====
// Add one byte to a record and increase size property appropriately.

SPEAddByteToRecord(bits(8) b)
    assert SPERecordSize < SPEMaxRecordSize;
    SPERecordData[SPERecordSize] = b;
    SPERecordSize = SPERecordSize + 1;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEAddPacketToRecord

```
// SPEAddPacketToRecord()
// =====
// Add passed header and payload data to the record.
// Payload must be a multiple of 8.

SPEAddPacketToRecord(bits(2) header_hi, bits(4) header_lo,
                     bits(N) payload)
    assert N MOD 8 == 0;
    bits(2) sz;
    case N of
        when 8 sz = '00';
        when 16 sz = '01';
        when 32 sz = '10';
        when 64 sz = '11';
        otherwise Unreachable();

    constant bits(8) header = header_hi:sz:header_lo;
    SPEAddByteToRecord(header);
    for i = 0 to (N DIV 8)-1
        SPEAddByteToRecord(Elem[payload, i, 8]);
```



```

// SPEBranch()
// =====
// Called on every branch if SPE is present. Maintains previous branch target
// and branch related SPE functionality.

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag)
    constant boolean is_isb = FALSE;
    SPEBranch(target, branch_type, conditional, taken_flag, is_isb);

SPEBranch(bits(N) target, BranchType branch_type, boolean conditional, boolean taken_flag,
    boolean is_isb)
    // If the PE implements branch prediction, data about (mis)prediction is collected
    // through the PMU events.

    boolean collect_prev_br;
    constant boolean collect_prev_br_eret = boolean IMPLEMENTATION_DEFINED "SPE prev br on eret";
    constant boolean collect_prev_br_exception = (boolean IMPLEMENTATION_DEFINED
        "SPE prev br on exception");
    constant boolean collect_prev_br_isb = boolean IMPLEMENTATION_DEFINED "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;

    // Implements previous branch target functionality
    if (taken_flag && !IsZero(PMSIDR_EL1.PBT) && StatisticalProfilingEnabled() &&
        collect_prev_br) then

        if SPESampleInFlight then
            // Save the target address for it to be added to record.
            constant bits(64) previous_target = SPESamplePreviousBranchAddress;
            SPESampleAddress[SPEAddrPosPrevBranchTarget]<63:0> = previous_target<63:0>;
            constant boolean previous_branch_valid = SPESamplePreviousBranchAddressValid;
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = previous_branch_valid;
            SPESamplePreviousBranchAddress<55:0> = target<55:0>;

        bit ns;
        bit nse;
        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();

        SPESamplePreviousBranchAddress<63> = ns;
        SPESamplePreviousBranchAddress<60> = nse;
        SPESamplePreviousBranchAddress<62:61> = PSTATE.EL;
        SPESamplePreviousBranchAddressValid = TRUE;

    if !StatisticalProfilingEnabled() then
        if taken_flag then
            // Invalidate previous branch address, if profiling is disabled
            // or prohibited.
            SPESamplePreviousBranchAddressValid = FALSE;
        return;

    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
    SPESampleAddress[SPEAddrPosBranchTarget]<55:0> = target<55:0>;

bit ns;
bit nse;
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();

SPESampleAddress[SPEAddrPosBranchTarget]<63> = ns;
SPESampleAddress[SPEAddrPosBranchTarget]<60> = nse;
SPESampleAddress[SPEAddrPosBranchTarget]<62:61> = PSTATE.EL;
SPESampleAddressValid[SPEAddrPosBranchTarget] = TRUE;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEBufferIsFull

```

// SPEBufferIsFull()
// =====
// Return true if another full size sample record would not fit in the
// profiling buffer.

boolean SPEBufferIsFull()
    constant integer write_pointer_limit = UInt(PMBLIMITR_EL1.LIMIT;Zeros(12));
    constant integer current_write_pointer = UInt(PMBPTR_EL1);
    constant integer record_max_size = 1<<UInt(PMSIDR_EL1.MaxSize);
    return current_write_pointer > (write_pointer_limit - record_max_size);

```



```

// SPECcollectRecord()
// =====
// Returns TRUE if the sampled class of instructions or operations, as
// determined by PMSFCR_EL1, are recorded and FALSE otherwise.

boolean SPECcollectRecord(bits(64) events, integer total_latency, SPEOpType optype)

    // "mask" defines which Events packet bits are checked by the filter
    bits(64) mask = Zeros(64);
    constant bits(64) impdef_mask = bits(64) IMPLEMENTATION_DEFINED "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
    if IsFeatureImplemented(FEAT_SPEvlp4) then
        mask<23:19> = '11111'; // Snoop hit, recent fetch, modified,
                                // L2 access, L2 hit
    else
        mask<31:24> = impdef_mask<31:24>;
    if IsFeatureImplemented(FEAT_SPEvlp1) && IsFeatureImplemented(FEAT_SVE) then
        mask<18:17> = '11'; // Predicates
    if IsFeatureImplemented(FEAT_TME) then
        mask<16> = '1'; // Transactional state
    mask<15:12> = impdef_mask<15:12>;
    if IsFeatureImplemented(FEAT_SPEvlp1) then
        mask<11> = '1'; // Data alignment
    if IsFeatureImplemented(FEAT_SPEvlp4) then
        mask<10:8> = '111'; // Remote access, LLC access, LLC miss
    else
        mask<10:8> = impdef_mask<10:8>;
    mask<7> = '1'; // Mispredicted
    if IsFeatureImplemented(FEAT_SPEvlp2) then
        mask<6> = '1'; // Not taken
    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>;

    constant bits(64) e = events AND mask;

    // Filtering by event
    constant bits(64) evfr = PMSEVFR_EL1 AND mask;
    boolean is_rejected_event = FALSE;
    constant boolean is_evt = IsZero(NOT(e) AND evfr);
    if PMSFCR_EL1.FE == '1' then
        // Filtering by event is enabled
        if !IsZero(evfr) then
            // Not UNPREDICTABLE case
            is_rejected_event = !is_evt;
        else
            is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

    // Filtering by inverse event
    boolean is_rejected_nevent = FALSE;
    boolean is_nevt;
    if IsFeatureImplemented(FEAT_SPEvlp2) then
        constant bits(64) nevfr = 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 UNPREDICTABLE case
                is_rejected_nevent = !is_nevt;
            else
                is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    else
        is_nevt = TRUE; // not implemented

```



```

if (IsFeatureImplemented(FEAT_SPEv1p2) && PMSFCR_EL1.<FnE,FE> == '11' &&
    !IsZero(PMSEVFR_EL1 AND PMSNEVFR_EL1 AND mask)) then
    // UNPREDICTABLE case due to combination of filter and inverse filter
    is_rejected_nevent = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);
    is_rejected_event = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

if is_evt && is_nevt then
    PMUEvent(PMU_EVENT_SAMPLE_FEED_EVENT);

constant boolean is_rejected_type = SPEFilterByType(SPESampleOpAttr);

// Filter by latency
boolean is_rejected_latency = FALSE;
constant boolean is_lat = (total_latency < UInt(PMSLATFR_EL1.MINLAT));
if is_lat then PMUEvent(PMU_EVENT_SAMPLE_FEED_LAT);

if PMSFCR_EL1.FL == '1' then
    // Filtering by latency is enabled
    if !IsZero(PMSLATFR_EL1.MINLAT) then
        // Not an UNPREDICTABLE case
        is_rejected_latency = !is_lat;
    else
        is_rejected_latency = ConstrainUnpredictableBool(Unpredictable_BADPMSFCR);

// Filtering by Data Source
boolean is_rejected_data_source = FALSE;
if (IsFeatureImplemented(FEAT_SPE_FDS) && SPESampleDataSourceValid &&
    (SPESampleOpAttr.op_type IN {SPEOpType\_Load, SPEOpType\_LoadAtomic})) then
    constant bits(16) data_source = SPESampleDataSource;
    constant integer index = UInt(data_source<5:0>);
    constant boolean is_ds = PMSDSFR_EL1<index> == '1';
    if is_ds then PMUEvent(PMU_EVENT_SAMPLE_FEED_DS);
    if PMSFCR_EL1.FDS == '1' then
        // Filtering by Data Source is enabled
        is_rejected_data_source = !is_ds;

boolean return_value;
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);

return return_value;

```



```

// 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.

SPEConstructRecord()
    // Empty the record.
    SPEEmptyRecord\(\);

    // Add contextEL1 if available
    if SPESampleContextEL1Valid then
        SPEAddPacketToRecord('01', '0100', SPESampleContextEL1);

    // Add contextEL2 if available
    if SPESampleContextEL2Valid then
        SPEAddPacketToRecord('01', '0101', SPESampleContextEL2);

    // Add valid counters
    for counter_index = 0 to (SPEMaxCounters - 1)
        if SPESampleCounterValid[counter_index] then
            if counter_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':counter_index<4:3>);
            // Check for overflow
            constant boolean large_counters = boolean IMPLEMENTATION_DEFINED "SPE 16bit counters";
            if SPESampleCounter[counter_index] > 0xFFFF && large_counters then
                SPESampleCounter[counter_index] = 0xFFFF;
            elseif SPESampleCounter[counter_index] > 0xFFF then
                SPESampleCounter[counter_index] = 0xFFF;

            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '1':counter_index<2:0>,
                SPESampleCounter[counter_index]<15:0>);

    // Add valid addresses
    if IsFeatureImplemented(FEAT_SPEv1p2) then
        // Under the some conditions, it is IMPLEMENTATION_DEFINED whether
        // previous branch packet is present.
        constant boolean include_prev_br = (boolean IMPLEMENTATION_DEFINED
            "SPE get prev br if not br");
        if SPESampleOpAttr.op_type != SPEOpType\_Branch && !include_prev_br then
            SPESampleAddressValid[SPEAddrPosPrevBranchTarget] = FALSE;

    // 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;

    for address_index = 0 to (SPEMaxAddrs - 1)
        if SPESampleAddressValid[address_index] then
            if address_index >= 8 then
                // Need extended format
                SPEAddByteToRecord('001000':address_index<4:3>);
            // Add byte0 for short format (byte1 for extended format)
            SPEAddPacketToRecord('10', '0':address_index<2:0>,
                SPESampleAddress[address_index]);

    // Add Data Source
    if SPESampleDataSourceValid then
        constant integer ds_payload_size = SPEGetDataSourcePayloadSize();
        SPEAddPacketToRecord('01', '0011', SPESampleDataSource<8*ds_payload_size-1:0>);

    // Construct class and subclass
    constant bit cond = if SPESampleOpAttr.is_conditional then '1' else '0';
    constant bit fp = if SPESampleOpAttr.is_floating_point then '1' else '0';
    constant bit ldst = if SPESampleOpAttr.op_type == SPEOpType\_Store then '1' else '0';
    constant bit ar = if SPESampleOpAttr.is_acquire_release then '1' else '0';
    constant bit excl = if SPESampleOpAttr.is_exclusive then '1' else '0';
    constant bit at = if SPESampleOpAttr.at then '1' else '0';

```

```

constant bit indirect = if SPESampleOpAttr.branch_is_direct then '0' else '1';
constant bit pred = if SPESampleOpAttr.is_predicated then '1' else '0';
constant bit sg = if SPESampleOpAttr.is_gather_scatter then '1' else '0';
constant bits(3) evl = SPESampleOpAttr.evl;
constant bit simd = if SPESampleOpAttr.is_simd then '1' else '0';
constant bits(4) ets = 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'.
constant bit common = '1';

bits(2) op_class;
bits(8) op_subclass;
if SPESampleOpAttr.op_type == SPEOpType\_Other then
    op_class = '00';
    op_subclass = Zeros(5):simd:fp:cond;
elsif SPESampleOpAttr.op_type == SPEOpType\_OtherSVE then
    op_class = '00';
    op_subclass = '0':evl:'1':pred:fp:'0';
elsif SPESampleOpAttr.op_type == SPEOpType\_OtherSME then
    op_class = '00';
    op_subclass = '1':ets<3:1>:'1':ets<0>:fp:'0';
elsif SPESampleOpAttr.op_type == SPEOpType\_Branch then
    op_class = '10';
    bits(2) cr = '00';
    bit gcs = '0';
    if IsFeatureImplemented(FEAT_SPE_CRR) then
        if SPESampleOpAttr.branch_has_link then
            cr = '01';
        elsif SPESampleOpAttr.procedure_return then
            cr = '10';
        else
            cr = '11';
    if IsFeatureImplemented(FEAT_GCS) then
        if (SPESampleOpAttr.ldst_type == SPELDSTType\_GCS &&
            (SPESampleOpAttr.branch_has_link || SPESampleOpAttr.procedure_return)) then
            gcs = '1';
    op_subclass = Zeros(3):cr:gcs:indirect:cond;

elsif 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
            // GCSSS2 is converted to GCS, should not appear here
            Unreachable();
        otherwise
            Unreachable();
    else
        Unreachable();

```

```

// Add operation details
SPEAddPacketToRecord('01', '10':op_class, op_subclass);

// Add events
// Get size of payload in bytes.
constant integer payload_size = SPEGetEventsPayloadSize();

SPEAddPacketToRecord('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('01', '0001', SPESampleTimestamp);
else
    SPEAddByteToRecord('00000001');

// Add padding
while SPERecordSize MOD (1<<UInt(PMBIDR_EL1.Align)) != 0 do
    SPEAddByteToRecord(Zeros(8));
SPEWriteToBuffer();

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPECycle

```

// SPECycle()
// =====
// Function called at the end of every cycle. Responsible for asserting interrupts
// and advancing counters.

SPECycle()
    if !IsFeatureImplemented(FEAT_SPE) then
        return;

    // Increment pending counters
    if SPESampleInFlight then
        for i = 0 to (SPEMaxCounters - 1)
            if SPESampleCounterPending[i] then
                SPESampleCounter[i] = SPESampleCounter[i] + 1;

    // Assert PMBIRQ if appropriate.
    if SPEInterruptEnabled() then
        SetInterruptRequestLevel(InterruptID_PMBIRQ,
                                if PMBSR_EL1.S == '1' then Signal_High else Signal_Low);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEEmptyRecord

```

// SPEEmptyRecord()
// =====
// Reset record data.

SPEEmptyRecord()
    SPERecordSize = 0;
    for i = 0 to (SPEMaxRecordSize - 1)
        SPERecordData[i] = Zeros(8);

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEEncodeETS

```
// SPEEncodeETS()
// =====
// Encodes an integer tile size length into the ets field for the SPE operation type packet.

bits(4) SPEEncodeETS(integer size)
    bits(4) ets;
    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';
    return ets;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEEncodeEVL

```
// SPEEncodeEVL()
// =====
// Encodes an integer vector length into the evl field for the SPE operation type packet.

bits(3) SPEEncodeEVL(integer vl)
    bits(3) evl;
    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();
    return evl;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEEvent

```
// SPEEvent()
// =====
// Called by PMUEvent if a sample is in flight.
// Sets appropriate bit in SPESampleStorage.events.

SPEEvent(bits(16) pmuevent)
  case pmuevent of
    when PMU_EVENT_DSNP_HIT_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<23> = '1';
    when PMU_EVENT_L1D_LFB_HIT_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<22> = '1';
    when PMU_EVENT_L2D_LFB_HIT_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<22> = '1';
    when PMU_EVENT_L3D_LFB_HIT_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<22> = '1';
    when PMU_EVENT_LL_LFB_HIT_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<22> = '1';
    when PMU_EVENT_L1D_CACHE_HITM_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<21> = '1';
    when PMU_EVENT_L2D_CACHE_HITM_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<21> = '1';
    when PMU_EVENT_L3D_CACHE_HITM_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<21> = '1';
    when PMU_EVENT_LL_CACHE_HITM_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<21> = '1';
    when PMU_EVENT_L2D_CACHE_LMISS_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<20> = '1';
    when PMU_EVENT_L2D_CACHE_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<19> = '1';
    when PMU_EVENT_SVE_PRED_EMPTY_SPEC
      if IsFeatureImplemented(FEAT_SPEvlp1) then
        SPESampleEvents<18> = '1';
    when PMU_EVENT_SVE_PRED_NOT_FULL_SPEC
      if IsFeatureImplemented(FEAT_SPEvlp1) then
        SPESampleEvents<17> = '1';
    when PMU_EVENT_LDST_ALIGN_LAT
      if IsFeatureImplemented(FEAT_SPEvlp1) then
        SPESampleEvents<11> = '1';
    when PMU_EVENT_REMOTE_ACCESS
      SPESampleEvents<10> = '1';
    when PMU_EVENT_LL_CACHE_MISS
      SPESampleEvents<9> = '1';
    when PMU_EVENT_LL_CACHE
      SPESampleEvents<8> = '1';
    when PMU_EVENT_BR_MIS_PRED
      SPESampleEvents<7> = '1';
    when PMU_EVENT_BR_MIS_PRED_RETIRE
      SPESampleEvents<7> = '1';
    when PMU_EVENT_DTLB_WALK
      SPESampleEvents<5> = '1';
    when PMU_EVENT_L1D_TLB
      SPESampleEvents<4> = '1';
    when PMU_EVENT_L1D_CACHE_REFILL
      if !IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<3> = '1';
    when PMU_EVENT_L1D_CACHE_LMISS_RD
      if IsFeatureImplemented(FEAT_SPEvlp4) then
        SPESampleEvents<3> = '1';
    when PMU_EVENT_L1D_CACHE
      SPESampleEvents<2> = '1';
    when PMU_EVENT_INST_RETIRE
      SPESampleEvents<1> = '1';
    when PMU_EVENT_EXC_TAKEN
      SPESampleEvents<0> = '1';
    otherwise return;
  return;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEFilterByType

```
// SPEFilterByType()
// =====
// Returns TRUE if the operation is to be discarded because of its OpAttrs, and FALSE otherwise.

boolean SPEFilterByType(SPEOpAttr opattr)
    // Bit order is {B,LD,ST,FP,SIMD}
    bits(5) flags, ctrl, mask;

    boolean is_load = FALSE;
    boolean is_store = FALSE;
    // 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.
    if (opattr.op_type IN {SPEOpType_Load, SPEOpType_LoadAtomic} ||
        (SPESampleOpAttr.ldst_type == SPELDSTType_GCS && opattr.procedure_return)) then
        // RET instruction with GCS should also count as Load
        is_load = TRUE;
    if (opattr.op_type IN {SPEOpType_Store, SPEOpType_LoadAtomic} ||
        (SPESampleOpAttr.ldst_type == SPELDSTType_GCS && opattr.branch_has_link)) then
        // BL instruction with GCS should also count as store.
        is_store = TRUE;
    // Construct flags bits
    // B
    flags<0> = (if opattr.op_type == SPEOpType_Branch then '1' else '0');
    // LD
    flags<1> = (if is_load then '1' else '0');
    // ST
    flags<2> = (if is_store then '1' else '0');
    // FP
    flags<3> = (if opattr.is_floating_point then '1' else '0');
    // SIMD
    flags<4> = (if opattr.is_simd then '1' else '0');
    if IsFeatureImplemented(FEAT_SPE_EFT) then
        ctrl = PMSFCR_EL1.<SIMD, FP, ST, LD, B>;
        mask = PMSFCR_EL1.<SIMDm, FPM, STm, LDm, Bm>;
    else
        ctrl = '00':PMSFCR_EL1.<ST, LD, B>;
        mask = '00000';

    constant bits(5) ctrl_or = (ctrl AND (NOT mask));
    constant bits(5) ctrl_and = (ctrl AND mask);

    constant boolean is_op = ((IsZero(ctrl_or) || !IsZero(flags AND ctrl_or)) &&
        ((flags AND mask) == ctrl_and));

    if flags<0> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_BR);
    if flags<1> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_LD);
    if flags<2> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_ST);
    if flags<3> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_FP);
    if flags<4> == '1' then PMUEvent(PMU_EVENT_SAMPLE_FEED_SIMD);

    boolean is_rejected_type = 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);
    if is_op && (!IsZero(ctrl) || !IsZero(mask)) then PMUEvent(PMU_EVENT_SAMPLE_FEED_OP);
    return is_rejected_type;
```



## 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.

boolean SPEFreezeOnEvent(integer idx)
    constant integer counters = 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;
    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;

    constant boolean freeze_on_event = PMBLIMITR_EL1.<E,PMFZ> == '11';
    constant boolean stopped = 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;
```

## 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.

(boolean, bits(16)) SPEGetDataSource(boolean is_load,
    AccessDescriptor accdesc,
    AddressDescriptor addrdesc);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetDataSourcePayloadSize

```
// SPEGetDataSourcePayloadSize()
// =====
// Returns the size of the Data Source payload in bytes.

integer SPEGetDataSourcePayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Data Source packet payload size";
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetEventsPayloadSize

```
// SPEGetEventsPayloadSize()
// =====
// Returns the size in bytes of the Events packet payload as an integer.

integer SPEGetEventsPayloadSize()
    return integer IMPLEMENTATION_DEFINED "SPE Events packet payload size";
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomBoolean

```
// SPEGetRandomBoolean()
// =====
// Returns a random or pseudo-random boolean value.

boolean SPEGetRandomBoolean();
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEGetRandomInterval

```
// SPEGetRandomInterval()
// =====
// Returns a random or pseudo-random byte for resetting COUNT or ECOUNT.

bits(8) SPEGetRandomInterval();
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEISB

```
// SPEISB()
// =====
// Called by ISB instruction, correctly calls SPEBranch to save previous branches.

SPEISB()
    constant bits(64) address = PC64 + 4;
    constant BranchType branch_type = BranchType\_DIR;
    constant boolean branch_conditional = FALSE;
    constant boolean taken = FALSE;
    constant boolean is_isb = TRUE;

    SPEBranch(address, branch_type, branch_conditional, taken, is_isb);
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEInterruptEnabled

```
// SPEInterruptEnabled()
// =====
// Return TRUE if the SPE interrupt request (PMBIRQ) is enabled, FALSE otherwise.

boolean SPEInterruptEnabled()
    return EffectivePMSCR\_EL1\_EE() == '00';
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPELDSTType

```
// SPELDSTType
// =====
// Type of a load or store operation.

enumeration SPELDSTType {
    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/SPEMaxAddrs

```
constant integer SPEMaxAddrs = 32;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxCounters

```
constant integer SPEMaxCounters = 32;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEMaxRecordSize

```
constant integer SPEMaxRecordSize = 2048;
```

## 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.

boolean SPEMultiAccessSample()
    // 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.

    // Are loads allowed by filter?
    constant boolean loads_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.LD == '1';
    // Are stores allowed by filter?
    constant boolean stores_pass_filter = PMSFCR_EL1.FT == '1' && PMSFCR_EL1.ST == '1';

    boolean record_load;
    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\(\);
    return record_load;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEOpAttr

```
// SPEOpAttr
// =====
// Attributes of sampled operation filtered by SPECollectRecord().

type SPEOpAttr is (
    SPEOpType op_type,
    SPELDSTType ldst_type,
    boolean 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,
    bits(3) evl,
    boolean is_gather_scatter,
    boolean is_exclusive,
    bits(4) ets,
    boolean addr_valid
)
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEOpType

```
// SPEOpType
// =====
// Types of operation filtered by SPECollectRecord().

enumeration SPEOpType {
    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
};
```



```

// Order of addresses in sample storage and architectural index values
constant integer SPEAddrPosPCVirtual = 0;
constant integer SPEAddrPosBranchTarget = 1;
constant integer SPEAddrPosDataVirtual = 2;
constant integer SPEAddrPosDataPhysical = 3;
constant integer SPEAddrPosPrevBranchTarget = 4;
// Order of counters in sample storage and architectural index values
constant integer SPECounterPosTotalLatency = 0;
constant integer SPECounterPosIssueLatency = 1;
constant integer SPECounterPosTranslationLatency = 2;
constant integer SPECounterPosAltIssueLatency = 4;
// Sample in-flight flag
boolean SPESampleInFlight = FALSE;
// Globally declared variables which stores data about the instruction being
// sampled.

// Context storage
bits(32) SPESampleContextEL1;
boolean SPESampleContextEL1Valid;
bits(32) SPESampleContextEL2;
boolean SPESampleContextEL2Valid;
bits(64) SPESamplePreviousBranchAddress;
boolean SPESamplePreviousBranchAddressValid;
// Data source storage
bits(16) SPESampleDataSource;
boolean SPESampleDataSourceValid;
// OpAttr storage
SPEOpAttr SPESampleOpAttr;
// Timestamp storage
bits(64) SPESampleTimestamp;
boolean SPESampleTimestampValid;
// Event storage
bits(64) SPESampleEvents;

// SPEPostExecution()
// =====
// Called after every executed instruction.

SPEPostExecution()
    if SPESampleInFlight then
        SPESampleInFlight = FALSE;
        PMUEvent (PMU_EVENT_SAMPLE_FEED);

    // Stop any pending counters
    for counter_index = 0 to (SPEMaxCounters - 1)
        if SPESampleCounterPending[counter_index] then
            SPEStopCounter (counter_index);

    // Record any IMPLEMENTATION DEFINED events
    constant bits(64) impdef_events = bits(64) IMPLEMENTATION_DEFINED "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>;
    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;
    // Bit 16 encodes whether the sample was collected in Transactional state.
    if IsFeatureImplemented(FEAT_TME) then
        SPESampleEvents<16> = if TSTATE.depth > 0 then '1' else '0';
    SPESampleEvents<6> = if (SPESampleOpAttr.is_conditional &&
        !SPESampleOpAttr.cond_pass) then '1' else '0';

    boolean discard = FALSE;
    if IsFeatureImplemented(FEAT_SPEv1p2) then
        discard = PMBLIMITR_EL1.FM == '10';
    if SPECollectRecord (SPESampleEvents,
        SPESampleCounter[SPECounterPosTotalLatency],

```

```

        SPESampleOpAttr.op_type) && !discard then
    SPEConstructRecord();
    if SPEBufferIsFull() then
        constant bits(6) bsc = '000001';      // Buffer full event
        OtherSPEManagementEvent(bsc);
        PMUEvent(PMU_EVENT_SAMPLE_BUFFER_FULL);

    SPEResetSampleStorage();

// Counter storage
array [0..SPEMaxCounters-1] of integer SPESampleCounter;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterValid;

array [0..SPEMaxCounters-1] of boolean SPESampleCounterPending;

// Address storage
array [0..SPEMaxAddrs-1] of bits(64) SPESampleAddress;

array [0..SPEMaxAddrs-1] of boolean SPESampleAddressValid;

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEPreExecution

```

// SPEPreExecution()
// =====
// Called prior to execution, for all instructions.

SPEPreExecution()
    if StatisticalProfilingEnabled() then
        PMUEvent(PMU_EVENT_SAMPLE_POP);
        if SPEToCollectSample() then
            if !SPESampleInFlight then
                SPESampleInFlight = TRUE;

                // Start total latency and issue latency counters for SPE
                SPEStartCounter(SPECounterPosTotalLatency);
                SPEStartCounter(SPECounterPosIssueLatency);

                SPESampleAddContext();

                SPESampleAddAddressPCVirtual();

                // 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();
            else
                PMUEvent(PMU_EVENT_SAMPLE_COLLISION);
                constant bits(2) target_el = DefaultSPEEvent();
                PMBSR\_EL[target_el].COLL = '1';

                // 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);

```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEProfilingStopped

```

// SPEProfilingStopped()
// =====

boolean SPEProfilingStopped()
    boolean stopped = (PMBSR_EL1.S == '1');
    if IsFeatureImplemented(FEAT_SPE_EXC) then
        if HaveEL(EL3) && MDCR_EL3.PMSEE == '1x' then
            stopped = stopped || (PMBSR_EL3.S == '1');
        if EffectivePMSCR\_EL2\_EE() == '1x' then
            stopped = stopped || (PMBSR_EL2.S == '1');
    return stopped;

```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleCounter

```
// SPEResetSampleCounter()
// =====
// Reset PMSICR_EL1.Counter

SPEResetSampleCounter()
    PMSICR_EL1.COUNT<31:8> = PMSIRR_EL1.INTERVAL;
    if PMSIRR_EL1.RND == '1' && PMSIDR_EL1.ERnd == '0' then
        PMSICR_EL1.COUNT<7:0> = SPEGetRandomInterval\(\);
    else
        PMSICR_EL1.COUNT<7:0> = Zeros(8);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEResetSampleStorage

```
integer SPERecordSize;

// SPEResetSampleStorage()
// =====
// Reset all variables inside sample storage.

SPEResetSampleStorage()
    // Context values
    SPESampleContextEL1 = Zeros(32);
    SPESampleContextEL1Valid = FALSE;
    SPESampleContextEL2 = Zeros(32);
    SPESampleContextEL2Valid = FALSE;

    // Counter values
    for i = 0 to (SPEMaxCounters - 1)
        SPESampleCounter[i] = 0;
        SPESampleCounterValid[i] = FALSE;
        SPESampleCounterPending[i] = FALSE;

    // Address values
    for i = 0 to (SPEMaxAddrs - 1)
        SPESampleAddressValid[i] = FALSE;
        SPESampleAddress[i] = Zeros(64);

    // 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;

array [0..SPEMaxRecordSize-1] of bits(8) SPERecordData;
```



## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddAddressPCVirtual

```
// SPESampleAddAddressPCVirtual()
// =====
// Save the current PC address to sample storage.

SPESampleAddAddressPCVirtual()
    constant bits(64) this_address = ThisInstrAddr(64);
    SPESampleAddress[SPEAddrPosPCVirtual]<55:0> = this_address<55:0>;

    bit ns;
    bit nse;
    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();

    constant bits(2) el = PSTATE.EL;
    SPESampleAddress[SPEAddrPosPCVirtual]<63:56> = ns:el:nse:Zeros(4);
    SPESampleAddressValid[SPEAddrPosPCVirtual] = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddContext

```
// SPESampleAddContext()
// =====
// Save contexts to sample storage if appropriate.

SPESampleAddContext()
    if CollectContextIDR1() then
        SPESampleContextEL1 = CONTEXTIDR_EL1<31:0>;
        SPESampleContextEL1Valid = TRUE;
    if CollectContextIDR2() then
        SPESampleContextEL2 = CONTEXTIDR_EL2<31:0>;
        SPESampleContextEL2Valid = TRUE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleAddTimeStamp

```
// SPESampleAddTimeStamp()
// =====
// Save the appropriate type of timestamp to sample storage.

SPESampleAddTimeStamp()
    constant TimeStamp timestamp = CollectTimeStamp();
    case timestamp of
        when TimeStamp\_None
            SPESampleTimestampValid = FALSE;
        otherwise
            SPESampleTimestampValid = TRUE;
            SPESampleTimestamp = GetTimeStamp(timestamp);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleExtendedLoadStore

```
// SPESampleExtendedLoadStore()
// =====
// Sets the subclass of the operation type packet for
// extended load/store operations.

SPESampleExtendedLoadStore(boolean ar, boolean excl, boolean at, boolean is_load)
    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;
    else
        SPESampleOpAttr.op_type = SPEOpType\_Store;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGCSSS2

```
// SPESampleGCSSS2()
// =====
// Sets the subclass of the operation type packet for GCSSS2 load/store operations.

SPESampleGCSSS2()
    // GCSSS2 does a read and a write.
    constant boolean record_load = SPEMultiAccessSample();
    SPESampleOpAttr.op_type = if record_load then SPEOpType\_Load else SPEOpType\_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType\_GCSSS2;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleGeneralPurposeLoadStore

```
// SPESampleGeneralPurposeLoadStore()
// =====
// Sets the subclass of the operation type packet for general
// purpose load/store operations.

SPESampleGeneralPurposeLoadStore(boolean is_load)
    SPESampleOpAttr.ldst_type = SPELDSTType\_General;
    SPESampleOpAttr.op_type = if is_load then SPEOpType\_Load else SPEOpType\_Store;
```

## 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.

SPESampleLoadStore(boolean is_load, AccessDescriptor accdesc, AddressDescriptor addrdesc)
    // 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;

    boolean sample_access = 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.
        boolean sample_load;
        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;
    else
        if !SPESampleOpAttr.addr_valid then
            sample_access = TRUE;

    if sample_access then
        // Data access virtual address
        SPESetDataVirtualAddress(addrdesc.vaddress);

        // Data access physical address
        if CollectPhysicalAddress() then
            SPESetDataPhysicalAddress(addrdesc, accdesc);

        SPESampleOpAttr.addr_valid = TRUE;

    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;

    // 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;
    (SPESampleDataSourceValid, SPESampleDataSource) = SPEGetDataSource(is_load, accdesc, addrdesc);
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemCopy

```
// SPESampleMemCopy()
// =====
// Sets the subclass of the operation type packet for Memory Copy load/store
// operations.

SPESampleMemCopy()
    // MemCopy does a read and a write.
    constant boolean record_load = SPEMultiAccessSample\(\);
    SPESampleOpAttr.op_type = if record_load then SPEOpType\_Load else SPEOpType\_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType\_MemCopy;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleMemSet

```
// SPESampleMemSet()
// =====
// Callback used by memory set instructions to pass data back to the SPU.

SPESampleMemSet()
    SPESampleOpAttr.op_type = SPEOpType\_Store;
    SPESampleOpAttr.ldst_type = SPELDSTType\_MemSet;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOnExternalCoproprocessor

```
// SPESampleOnExternalCoproprocessor()
// =====
// Called when the sampled instruction is executed on an SMCU or external coprocessor, sets bit 25
// of the events packet to 0b1.

SPESampleOnExternalCoproprocessor()
    SPESampleEvents<25> = '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpOther

```
// SPESampleOpOther()
// =====
// Add other operation to sample storage.

SPESampleOpOther(boolean conditional, boolean cond_pass, boolean is_fp, boolean is_simd)
    SPESampleOpAttr.is_simd = is_simd;
    SPESampleOpOther(conditional, cond_pass, is_fp);

SPESampleOpOther(boolean conditional, boolean cond_pass, boolean is_fp)
    SPESampleOpAttr.cond_pass = cond_pass;
    SPESampleOpAttr.is_floating_point = is_fp;
    SPESampleOpOther(conditional);

SPESampleOpOther(boolean conditional)
    SPESampleOpAttr.is_conditional = conditional;
    SPESampleOpAttr.op_type = SPEOpType\_Other;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSMEArrayOther

```
// SPESampleOpSMEArrayOther()
// =====
// Sets the subclass of the operation type packet to Other, SME array.

SPESampleOpSMEArrayOther(boolean floating_point, integer size)
    // 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;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSVEOther

```
// SPESampleOpSVEOther()
// =====
// Callback used by SVE, Other operations to pass data back to the SPU.

SPESampleOpSVEOther(integer vl, boolean predicated, boolean floating_point)
    SPESampleOpAttr.is_predicated = predicated;
    SPESampleOpAttr.is_floating_point = floating_point;
    SPESampleOpAttr.evl = SPEEncodeEVL(vl);
    SPESampleOpAttr.op_type = SPEOpType\_OtherSVE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleOpSVESMELoadStore

```
// SPESampleOpSVESMELoadStore()
// =====
// Callback used by SVE or SME loads and stores to pass data to SPE.

SPESampleOpSVESMELoadStore(boolean is_gather_scatter, integer vl, boolean predicated,
                           boolean is_load)
    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;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESampleSIMDFPLoadStore

```
// SPESampleSIMDFPLoadStore()
// =====
// Sets the subclass of the operation type packet for SIMD & FP
// load store operations.

SPESampleSIMDFPLoadStore(boolean is_load, boolean scalar)
    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;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPESetDataPhysicalAddress

```
// SPESetDataPhysicalAddress()
// =====
// Called from SampleLoadStore() to save data physical packet.

SPESetDataPhysicalAddress(AddressDescriptor addrdesc, AccessDescriptor accdesc)
    bit ns;
    bit nse;
    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();

    if IsFeatureImplemented(FEAT_MTE2) then
        bits(4) lat;
        if accdesc.tagchecked then
            SPESampleAddress[SPEAddrPosDataPhysical]<62> = '1'; // CH
            lat = AArch64.LogicalAddressTag(addrdesc.vaddress);
        else
            // CH is reset to 0 on each new packet
            // If the access is Unchecked, this is an IMPLEMENTATION_DEFINED choice
            // between 0b0000 and the Physical Address Tag
            boolean zero_unchecked;
            zero_unchecked = boolean IMPLEMENTATION_DEFINED "SPE PAT for tag unchecked access zero";
            if !zero_unchecked then
                lat = AArch64.LogicalAddressTag(addrdesc.vaddress);
            else
                lat = Zeros(4);
            SPESampleAddress[SPEAddrPosDataPhysical]<59:56> = lat;

    constant bits(56) paddr = addrdesc.paddress.address;
    SPESampleAddress[SPEAddrPosDataPhysical]<56-1:0> = paddr;
    SPESampleAddress[SPEAddrPosDataPhysical]<63> = ns;
    SPESampleAddress[SPEAddrPosDataPhysical]<60> = nse;
    SPESampleAddressValid[SPEAddrPosDataPhysical] = TRUE;
```

## 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.

SPESetDataVirtualAddress(bits(64) vaddress)
    bit tbi;
    tbi = EffectiveTBI(vaddress, FALSE, PSTATE.EL);
    boolean non_tbi_is_zeros;
    non_tbi_is_zeros = boolean IMPLEMENTATION_DEFINED "SPE non-tbi tag is zero";
    if tbi == '1' || !non_tbi_is_zeros then
        SPESampleAddress[SPEAddrPosDataVirtual]<63:0> = vaddress<63:0>;
    else
        SPESampleAddress[SPEAddrPosDataVirtual]<63:56> = Zeros(8);
        SPESampleAddress[SPEAddrPosDataVirtual]<55:0> = vaddress<55:0>;
    SPESampleAddressValid[SPEAddrPosDataVirtual] = TRUE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPESStartCounter

```
// SPESStartCounter()
// =====
// Enables incrementing of the counter at the passed index when SPECycle is called.

SPESStartCounter(integer counter_index)
    assert counter_index < SPEMaxCounters;
    SPESampleCounterPending[counter_index] = TRUE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPESStopCounter

```
// SPESStopCounter()
// =====
// Disables incrementing of the counter at the passed index when SPECycle is called.

SPESStopCounter(integer counter_index)
    SPESampleCounterValid[counter_index] = TRUE;
    SPESampleCounterPending[counter_index] = FALSE;
```

### Library pseudocode for aarch64/debug/statisticalprofiling/SPEToCollectSample

```
// SPEToCollectSample()
// =====
// Returns TRUE if the instruction which is about to be executed should be
// sampled. Returns FALSE otherwise.

boolean SPEToCollectSample()
    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' && PMSIDR_EL1.ERnd == '1' then
                PMSICR_EL1.ECOUNT = SPEGetRandomInterval();
            else
                return TRUE;
        if UInt(PMSICR_EL1.ECOUNT) != 0 then
            PMSICR_EL1.ECOUNT = PMSICR_EL1.ECOUNT - 1;
            if IsZero(PMSICR_EL1.ECOUNT) then
                return TRUE;
    return FALSE;
```

## Library pseudocode for aarch64/debug/statisticalprofiling/SPEWriteToBuffer

```
// SPEWriteToBuffer()
// =====
// Write the active record to the Profiling Buffer.

SPEWriteToBuffer()
    assert ProfilingBufferEnabled();

    // Check alignment
    constant integer align = UInt(PMBIDR_EL1.Align);
    constant boolean aligned = IsAligned(PMBPTR_EL1.PTR, 2^align);
    boolean ttw_fault_as_external_abort;
    ttw_fault_as_external_abort = boolean IMPLEMENTATION_DEFINED "SPE TTW fault External abort";

    FaultRecord fault;
    PhysMemRetStatus memstatus;
    AddressDescriptor addrdesc;
    AccessDescriptor accdesc;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = ProfilingBufferOwner();
    accdesc = CreateAccDescSPE(owning_ss, owning_el);

    constant bits(64) start_vaddr = PMBPTR_EL1<63:0>;
    for i = 0 to SPERecordSize - 1
        // If a previous write did not cause an issue
        if !SPEProfilingStopped() then
            (memstatus, addrdesc) = DebugMemWrite(PMBPTR_EL1<63:0>, accdesc, aligned,
                SPERecordData[i]);

            fault = addrdesc.fault;

            boolean ttw_fault;
            ttw_fault = fault.statuscode IN {Fault\_SyncExternalOnWalk, Fault\_SyncParityOnWalk};

            if IsFault(fault.statuscode) && !(ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteFault(PMBPTR_EL1<63:0>, fault);
            elseif IsFault(memstatus) || (ttw_fault && ttw_fault_as_external_abort) then
                DebugWriteExternalAbort(memstatus, addrdesc, start_vaddr);

            // Move pointer if no Buffer Management Event has been caused.
            if !SPEProfilingStopped() then
                PMBPTR_EL1 = PMBPTR_EL1 + 1;

    return;
```



## Library pseudocode for aarch64/debug/statisticalprofiling/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the current EL, FALSE otherwise.

boolean StatisticalProfilingEnabled()
    return StatisticalProfilingEnabled(PSTATE.EL);

// StatisticalProfilingEnabled()
// =====
// Return TRUE if Statistical Profiling is Enabled in the specified EL, FALSE otherwise.

boolean StatisticalProfilingEnabled(bits(2) el)
    if !IsFeatureImplemented(FEAT_SPE) || UsingAArch32() || !ProfilingBufferEnabled() then
        return FALSE;

    tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
    (owning_ss, owning_el) = ProfilingBufferOwner();
    if (UInt(owning_el) < UInt(el) || (tge_set && owning_el == EL1) ||
        owning_ss != SecurityStateAtEL(el)) then
        return FALSE;
    bit spe_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);

    return spe_bit == '1';
```

## Library pseudocode for aarch64/debug/statisticalprofiling/TimeStamp

```
// TimeStamp
// =====

enumeration TimeStamp {
    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
```



```

// AArch64.TakeExceptionInDebugState()
// =====
// Take an exception in Debug state to an Exception level using AArch64.

AArch64.TakeExceptionInDebugState(bits(2) target_el, ExceptionRecord exception_in)
    assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
    assert target_el != EL3 || EDSCR.SDD == '0';
    ExceptionRecord except = exception_in;
    boolean sync_errors;
    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);
        // The Effective value of SCTLR[].IESB might be zero in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable\_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_EL[target_el].EIS == '1' then
        // Synchronize the context, including Instruction Fetch Barrier effect
        SynchronizeContext();

    // If coming from AArch32 state, the top parts of the X[] registers might be set to zero
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);

    AArch64.ReportException(except, target_el);

    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0'; // Effective value of GCSCR_ELx.EXLOCKEN is 0 in Debug state

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR\_ELx[] = bits(64) UNKNOWN;
    ELR\_ELx[] = bits(64) UNKNOWN;

    // 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> = bits(5) UNKNOWN;
    PSTATE.IL = '0';
    if from_32 then // Coming from AArch32
        PSTATE.IT = '00000000';
        PSTATE.T = '0'; // PSTATE.J is RES0
    if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
        (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR_ELx[].SPAN == '0') then
        PSTATE.PAN = '1';
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
    if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = bit UNKNOWN;
    if IsFeatureImplemented(FEAT_SEBEP) then PSTATE.PPEND = '0';
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    EDSCR.ERR = '1';
    UpdateEDSCRFields(); // Update EDSCR processor state flags.

    if sync_errors then
        SynchronizeErrors();

```

```
EndOfInstruction();
```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointByteMatch

```
// AArch64.WatchpointByteMatch()
// =====

boolean AArch64.WatchpointByteMatch(integer n, bits(64) vaddress)
    constant AddressSize dbgtop = DebugAddrTop();
    constant integer cmpbottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3; // Word or doubleword
    bottom = cmpbottom;
    constant integer select = UInt(vaddress<cmpbottom-1:0>);
    byte_select_match = (DBGWCR_EL1[n].BAS<select> != '0');
    mask = 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
        LSB = (DBGWCR_EL1[n].BAS AND NOT(DBGWCR_EL1[n].BAS - 1)); MSB = (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

    // If the address mask is set to a reserved value, the behavior is CONSTRAINED UNPREDICTABLE.
    if mask > 0 && mask <= 2 then
        Constraint c;
        (c, 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

    // 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.
    constant boolean unpredictable_ress = (dbgtop < 55 && !IsOnes(DBGWVR_EL1[n]<63:dbgtop>) &&
        !IsZero(DBGWVR_EL1[n]<63:dbgtop>) &&
        ConstrainUnpredictableBool(Unpredictable\_DBGxVR\_RESS));
    constant integer cmpmsb = if unpredictable_ress then 63 else dbgtop;
    constant integer cmpls = if mask > bottom then mask else bottom;
    constant integer bottombit = bottom;
    boolean WVR_match = (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);

    return (WVR_match && byte_select_match);
```

## Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

WatchpointInfo AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size,
                                         AccessDescriptor accdesc)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    constant boolean enabled          = IsWatchpointEnabled(n);
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;
    lbnx = if IsFeatureImplemented(FEAT_Debugv8p9) then DBGWCR_EL1[n].LBNX else '00';
    linked_n = UInt(lbnx : DBGWCR_EL1[n].LBN);
    ssce = if IsFeatureImplemented(FEAT_RME) then DBGWCR_EL1[n].SSCE else '0';
    mismatch = IsFeatureImplemented(FEAT_BWE2) && DBGWCR_EL1[n].WT2 == '1';
    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, ssce, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, linked_n, isbreakpnt, PC64, accdesc);

    boolean ls_match;
    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;

    boolean value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, vaddress + byte);

    WatchpointInfo watchptinfo;
    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;
    return watchptinfo;
```

## Library pseudocode for aarch64/debug/watchpoint/DataCacheWatchpointSize

```
// DataCacheWatchpointSize
// =====
// Return the IMPLEMENTATION DEFINED data cache watchpoint size

integer DataCacheWatchpointSize()
    constant integer size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert IsPow2(size) && size >= 2^(UInt(CTR_EL0.DminLine) + 2) && size <= 2048;
    return size;
```

## Library pseudocode for aarch64/debug/watchpoint/IsWatchpointEnabled

```
// IsWatchpointEnabled()
// =====
// Returns TRUE if the effective value of DBGWCR_EL1[n].E is '1', and FALSE otherwise.

boolean IsWatchpointEnabled(integer n)
    if (n > 15 &&
        ((!HaltOnBreakpointOrWatchpoint() && !SelfHostedExtendedBPWPEEnabled()) ||
         (HaltOnBreakpointOrWatchpoint() && EDSCR2.EHBWE == '0'))) then
        return FALSE;
    return DBGWCR_EL1[n].E == '1';
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(FaultRecord fault)

    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);
        else
            AArch64.WatchpointException(fault);
    elseif fault.gpcf.gpf != GPCF\_None && ReportAsGPCEException(fault) then
        TakeGPCEException(fault);
    elseif fault.statuscode == Fault\_TagCheck then
        AArch64.RaiseTagCheckFault(fault);
    elseif fault.accessdesc.acctype == AccessType\_IFETCH then
        AArch64.InstructionAbort(fault);
    else
        AArch64.DataAbort(fault);
```

## 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.

ExceptionRecord AArch64.AbortSyndrome(Exception exceptype, FaultRecord fault, bits(2) target_el)
    except = ExceptionSyndrome(exceptype);

    if (!IsFeatureImplemented(FEAT_PFAR) ||
        !IsExternalSyncAbort(fault) ||
        (EL2Enabled() && HCR_EL2.VM == '1' && target_el == EL1)) then
        except.pavalid = FALSE;
    else
        except.pavalid = boolean IMPLEMENTATION_DEFINED "PFAR_ELx is valid";

    except.syndrome = AArch64.FaultSyndrome(exceptype, fault, except.pavalid);
    if fault.statuscode == Fault\_TagCheck then
        if IsFeatureImplemented(FEAT_MTE4) then
            except.vaddress = ZeroExtend(fault.vaddress, 64);
        else
            except.vaddress = bits(4) UNKNOWN : fault.vaddress<59:0>;
    else
        except.vaddress = ZeroExtend(fault.vaddress, 64);

    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;

    return except;
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()
    constant bits(64) pc = ThisInstrAddr(64);

    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====

AArch64.DataAbort(FaultRecord fault)
    bits(2) target_el;
    if IsExternalAbort(fault) then
        target_el = SyncExternalAbortTarget(fault);
    else
        route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
            (HCR_EL2.TGE == '1' ||
            (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf == GPCF\_Fail &&
            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;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant boolean route_to_serr = (IsExternalAbort(fault) &&
        AArch64.RouteToSErrorOffset(target_el));
    constant integer vect_offset = if route_to_serr then 0x180 else 0x0;

    ExceptionRecord except;
    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);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## 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.

TCFType AArch64.EffectiveTCF(bits(2) el, boolean read)
    bits(2) tcf;

    constant 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();

    if tcf == '11' then          // Reserved value
        if !IsFeatureImplemented(FEAT_MTE_ASYM_FAULT) then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable\_RESTCF, 2);

    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;
        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;
            else
                // Otherwise, Tag Check Faults have no effect on the PE
                return TCFType\_Ignore;
        otherwise
            Unreachable();
```



## Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(FaultRecord fault)
// External aborts on instruction fetch must be taken synchronously
if IsFeatureImplemented(FEAT_DoubleFault) then
    assert fault.statuscode != Fault\_AsyncExternal;

bits(2) target_el;
if IsExternalAbort(fault) then
    target_el = SyncExternalAbortTarget(fault);
else
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' ||
        (IsFeatureImplemented(FEAT_RME) && fault.gpcf.gpf == GPCF\_Fail &&
        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;

constant bits(64) preferred_exception_return = ThisInstrAddr(64);
integer vect_offset;

if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
    vect_offset = 0x180;
else
    vect_offset = 0x0;

constant ExceptionRecord except = AArch64.AbortSyndrome(Exception\_InstructionAbort, fault,
    target_el);
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.PCAlignmentFault

```
// AArch64.PCAlignmentFault()
// =====
// Called on unaligned program counter in AArch64 state.

AArch64.PCAlignmentFault()

constant bits(64) preferred_exception_return = ThisInstrAddr(64);
vect_offset = 0x0;

except = ExceptionSyndrome(Exception\_PCAlignment);
except.vaddress = ThisInstrAddr(64);
bits(2) target_el = EL1;
if UInt(PSTATE.EL) > UInt(EL1) then
    target_el = PSTATE.EL;
elseif EL2Enabled() && HCR_EL2.TGE == '1' then
    target_el = EL2;
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// =====
// Raise a Tag Check Fault exception.

AArch64.RaiseTagCheckFault(FaultRecord fault)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    bits(2) target_el = 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;

    except = AArch64.AbortSyndrome(Exception DataAbort, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =====
// Records a Tag Check Fault exception into the appropriate TFSR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    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';
        when EL1 if ttbr == '0' then TFSR_EL1.TF0 = '1'; else TFSR_EL1.TF1 = '1';
        when EL0 if ttbr == '0' then TFSR_EL0.TF0 = '1'; else TFSR_EL0.TF1 = '1';
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.RouteToSErrorOffset

```
// AArch64.RouteToSErrorOffset()
// =====
// Returns TRUE if synchronous External abort exceptions are taken to the
// appropriate SError vector offset, and FALSE otherwise.

boolean AArch64.RouteToSErrorOffset(bits(2) target_el)
    if !IsFeatureImplemented(FEAT_DoubleFault) then return FALSE;

    bit ease_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';
        when EL1
            if IsFeatureImplemented(FEAT_DoubleFault2) && IsSCTLR2EL1Enabled() then
                ease_bit = SCTLR2_EL1.EASE;
            else
                ease_bit = '0';
    return (ease_bit == '1');
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_SPAlignment);

    bits(2) target_el = EL1;
    if UInt(PSTATE.EL) > UInt(EL1) then
        target_el = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        target_el = EL2;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/AArch64.TagCheckFault

```
// AArch64.TagCheckFault()
// =====
// Handle a Tag Check Fault condition.

AArch64.TagCheckFault(bits(64) vaddress, AccessDescriptor accdesc)
    constant TCFType tcftype = AArch64.EffectiveTCF(accdesc.el, accdesc.read);

    case tcftype of
        when TCFType\_Sync
            FaultRecord fault = NoFault();
            fault.accessdesc = accdesc;
            fault.write = accdesc.write;
            fault.statuscode = Fault\_TagCheck;
            fault.vaddress = vaddress;
            AArch64.RaiseTagCheckFault(fault);
        when TCFType\_Async
            AArch64.ReportTagCheckFault(accdesc.el, vaddress<55>);
        when TCFType\_Ignore
            return;
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_BranchTarget);
    except.syndrome.iss<1:0> = PSTATE.BTYPE;
    except.syndrome.iss<24:2> = Zeros(23); // RES0

    bits(2) target_el = 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;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/aborts/TCFType

```
// TCFType
// =====

enumeration TCFType { TCFType_Sync, TCFType_Async, TCFType_Ignore };
```



```

// TakeGPCEException()
// =====
// Report Granule Protection Exception faults

TakeGPCEException(FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert IsFeatureImplemented(FEAT_LSE);
    assert IsFeatureImplemented(FEAT_HAFDBS);
    assert IsFeatureImplemented(FEAT_DoubleFault);

    ExceptionRecord except;

    except.exceptype = Exception\_GPC;
    except.vaddress = ZeroExtend(fault.vaddress, 64);
    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;

    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

    // 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
    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
    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

    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\_HWUpdateAccessFlag, Fault\_Exclusive} then
        except.syndrome.iss<6> = bit UNKNOWN; // WnR
    elseif fault.accessdesc.atomicop && IsExternalAbort(fault) then
        except.syndrome.iss<6> = bit UNKNOWN; // WnR
    else
        except.syndrome.iss<6> = if fault.write then '1' else '0'; // WnR

    except.syndrome.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level); // xFSC

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant bits(2) target_el = EL3;

    integer vect_offset;
    if IsExternalAbort(fault) && AArch64.RouteToSErrorOffset(target_el) then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

```

```
AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/async/AArch64.TakeDelegatedSErrorException

```
// AArch64.TakeDelegatedSErrorException()
// =====

AArch64.TakeDelegatedSErrorException()
    assert IsFeatureImplemented(FEAT_E3DSE) && PSTATE.EL != EL3 && SCR_EL3.<EndDSE,DSE> == '11';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception\_SError);

    bits(2) target_el;
    boolean dsei_masked;
    (dsei_masked, target_el) = AArch64.DelegatedSErrorTarget();
    assert !dsei_masked;
    except.syndrome.iss<24> = VESR_EL3.IDS;
    except.syndrome.iss<23:0> = VESR_EL3.ISS;
    ClearPendingDelegatedSError();

    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

### Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalFIQException

```
// AArch64.TakePhysicalFIQException()
// =====

AArch64.TakePhysicalFIQException()

    route_to_el3 = HaveEL(EL3) && SCR_EL3.FIQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.FMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;
    except = 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);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalIRQException

```
// AArch64.TakePhysicalIRQException()
// =====
// Take an enabled physical IRQ exception.

AArch64.TakePhysicalIRQException()

    route_to_el3 = HaveEL\(EL3\) && SCR_EL3.IRQ == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
                    (HCR_EL2.TGE == '1' || HCR_EL2.IMO == '1'));
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = 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);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(boolean implicit_esb)
    boolean masked;
    bits(2) target_el;

    (masked, target_el) = PhysicalSErrorTarget();
    assert !masked;

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x180;
    except = ExceptionSyndrome(Exception\_SError);
    constant bits(25) syndrome = AArch64.PhysicalSErrorSyndrome(implicit_esb);

    if IsSErrorEdgeTriggered() then
        ClearPendingPhysicalSError();

    except.syndrome.iss = syndrome;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualFIQException

```
// AArch64.TakeVirtualFIQException()
// =====

AArch64.TakeVirtualFIQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.FMO == '1'; // Virtual IRQ enabled if TGE==0 and FMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x100;

    except = ExceptionSyndrome(Exception\_FIQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```



## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualIRQException

```
// AArch64.TakeVirtualIRQException()
// =====

AArch64.TakeVirtualIRQException()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.IMO == '1'; // Virtual IRQ enabled if TGE==0 and IMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x80;

    except = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException()

    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();
    assert HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1'; // Virtual SError enabled if TGE==0 and AMO==1

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x180;
    except = ExceptionSyndrome(Exception_SError);

    if IsFeatureImplemented(FEAT_RAS) then
        except.syndrome.iss<24> = VSESR_EL2.IDS;
        except.syndrome.iss<23:0> = VSESR_EL2.ISS;
    else
        constant bits(25) syndrome = bits(25) IMPLEMENTATION_DEFINED "Virtual SError syndrome";
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then except.syndrome.iss = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.BreakpointException

```
// AArch64.BreakpointException()
// =====

AArch64.BreakpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_Breakpoint, fault, target_el);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareBreakpoint

```
// AArch64.SoftwareBreakpoint()
// =====

AArch64.SoftwareBreakpoint(bits(16) immediate)

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} &&
                    EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = 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);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.SoftwareStepException

```
// AArch64.SoftwareStepException()
// =====

AArch64.SoftwareStepException()
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = 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';
    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);
```

## 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.

AArch64.VectorCatchException(FaultRecord fault)
    assert PSTATE.EL != EL2;
    assert EL2Enabled() && (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1');

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    except = AArch64.AbortSyndrome(Exception_VectorCatch, fault, EL2);

    AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    bits(2) target_el;
    vect_offset = 0x0;
    target_el = if (PSTATE.EL == EL2 || route_to_el2) then EL2 else EL1;

    ExceptionRecord except;
    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);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ExceptionClass

```
// AArch64.ExceptionClass()
// =====
// Returns the Exception Class and Instruction Length fields to be reported in ESR

(integer,bit) AArch64.ExceptionClass(Exception exceptype, bits(2) target_el)

    il_is_valid = TRUE;
    from_32 = UsingAArch32();
    integer ec;
    case exceptype of
        when Exception Uncategorized          ec = 0x00; il_is_valid = FALSE;
        when Exception WxTrap                  ec = 0x01;
        when Exception CP15RTTTrap              ec = 0x03; assert from_32;
        when Exception CP15RRTTrap              ec = 0x04; assert from_32;
        when Exception CP14RTTTrap              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 LDST64BTrap              ec = 0x0A;
        when Exception TSTARTAccessTrap        ec = 0x1B;
        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 = 0x11;
        when Exception HypervisorCall          ec = 0x12;
        when Exception MonitorCall            ec = 0x13;
        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 = 0x20; il_is_valid = FALSE;
        when Exception PCAlignment            ec = 0x22; il_is_valid = FALSE;
        when Exception DataAbort              ec = 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 = 0x28;
        when Exception SError                  ec = 0x2F; il_is_valid = FALSE;
        when Exception Breakpoint              ec = 0x30; il_is_valid = FALSE;
        when Exception SoftwareStep            ec = 0x32; il_is_valid = FALSE;
        when Exception Watchpoint             ec = 0x34; il_is_valid = FALSE;
        when Exception NV2Watchpoint          ec = 0x35; il_is_valid = FALSE;
        when Exception SoftwareBreakpoint     ec = 0x38;
        when Exception VectorCatch            ec = 0x3A; il_is_valid = FALSE; assert from_32;
        when Exception Profiling              ec = 0x3D;
        otherwise                               Unreachable();

    if ec IN {0x20,0x24,0x30,0x32,0x34} && target_el == PSTATE.EL then
        ec = ec + 1;

    if ec IN {0x11,0x12,0x13,0x28,0x38} && !from_32 then
        ec = ec + 4;

    bit il;
    if il_is_valid then
        il = if ThisInstrLength() == 32 then '1' else '0';
    else
        il = '1';
    assert from_32 || il == '1'; // AArch64 instructions always 32-bit

    return (ec,il);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.ReportException

```
// AArch64.ReportException()
// =====
// Report syndrome information for exception taken to AArch64 state.

AArch64.ReportException(ExceptionRecord except, bits(2) target_el)

    constant Exception exceptype = except.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = except.syndrome.iss;
    iss2 = 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';

    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] = bits(64) UNKNOWN;

    if except.ipavalid then
        HPFAR_EL2<47:4> = except.ipaddress<55:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS\_Secure then
            HPFAR_EL2.NS = except.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<47:4> = bits(44) UNKNOWN;

    if except.pavalid then
        bits(64) faultaddr = ZeroExtend(except.paddress.address, 64);
        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';
            if exceptype == Exception\_GPC then
                faultaddr<11:0> = Zeros(12);
            else
                faultaddr<63> = if except.paddress.paspace == PAS\_NonSecure then '1' else '0';
                PFAR\_EL[target_el] = faultaddr;
        elsif (IsFeatureImplemented(FEAT_PFAR) ||
            (IsFeatureImplemented(FEAT_RME) && target_el == EL3)) then
            PFAR\_EL[target_el] = bits(64) UNKNOWN;
    return;
```

## 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.  
  
AArch64.ResetControlRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/exceptions/exceptions/AArch64.TakeReset

```
// AArch64.TakeReset()
// =====
// Reset into AArch64 state

AArch64.TakeReset(boolean cold_reset)
    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;

    // 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
    if IsFeatureImplemented(FEAT_SME) then
        PSTATE.<SM,ZA> = '00'; // Disable Streaming SVE mode & ZA storage
        ResetSMEState('0');
    if IsFeatureImplemented(FEAT_SSBS) then
        PSTATE.SSBS = bit IMPLEMENTATION_DEFINED "PSTATE.SSBS bit at reset";
    if IsFeatureImplemented(FEAT_GCS) then
        PSTATE.EXLOCK = '0'; // PSTATE.EXLOCK is reset to 0 when resetting into AArch64
    if IsFeatureImplemented(FEAT_UINJ) then
        PSTATE.UINJ = '0'; // PSTATE.UINJ is reset to 0 when resetting into AArch64
    PSTATE.IL = '0'; // Clear Illegal Execution state bit

    if IsFeatureImplemented(FEAT_TME) then TSTATE.depth = 0; // Non-transactional state

    // 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();
        AArch64.ResetSpecialRegisters();
        ResetExternalDebugRegisters(cold_reset);

    bits(64) rv; // IMPLEMENTATION DEFINED reset vector

    if HaveEL\(EL3\) then
        rv = RVBAR_EL3;
    elsif HaveEL\(EL2\) then
        rv = RVBAR_EL2;
    else
        rv = RVBAR_EL1;

    // The reset vector must be correctly aligned
    constant AddressSize pamax = AArch64.PAMax();
    assert IsZero(rv<63:pamax>) && IsZero(rv<1:0>);

    constant boolean branch_conditional = FALSE;
    EDPRSR.R = '0'; // Leaving Reset State.
    BranchTo(rv, BranchType\_RESET, branch_conditional);
```

## Library pseudocode for aarch64/exceptions/ieeeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    except = ExceptionSyndrome(Exception\_FPTrappedException);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            except.syndrome.iss<23> = '1'; // TFV
        else
            except.syndrome.iss<23> = '0'; // TFV
    else
        except.syndrome.iss<23> = '1'; // TFV
    except.syndrome.iss<10:8> = bits(3) UNKNOWN; // 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> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled() && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 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);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallHypervisor

```
// AArch64.CallHypervisor()
// =====
// Performs a HVC call

AArch64.CallHypervisor(bits(16) immediate)
    assert HaveEL(EL2);

    if UsingAArch32() then AArch32.ITAdvance();
    SSAdvance();
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_HypervisorCall);
    except.syndrome.iss<15:0> = immediate;

    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```



## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSecureMonitor

```
// AArch64.CallSecureMonitor()
// =====

AArch64.CallSecureMonitor(bits(16) immediate)
    assert HaveEL\(EL3\) && !ELUsingAArch32\(EL3\);
    if UsingAArch32\(\) then AArch32.ITAdvance\(\);
    HSAdvance\(\);
    SSAdvance\(\);
    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome\(Exception\_MonitorCall\);
    except.syndrome.iss<15:0> = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    AArch64.TakeException\(EL3, except, preferred\_exception\_return, vect\_offset\);
```

## Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate)
    if UsingAArch32\(\) then AArch32.ITAdvance\(\);
    SSAdvance\(\);
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled\(\) && HCR_EL2.TGE == '1';

    constant bits(64) preferred_exception_return = NextInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome\(Exception\_SupervisorCall\);
    except.syndrome.iss<15:0> = immediate;
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
    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\(EL1, except, preferred\_exception\_return, vect\_offset\);
```



```

// AArch64.TakeException()
// =====
// Take an exception to an Exception level using AArch64.

AArch64.TakeException(bits(2) target_el, ExceptionRecord exception_in,
                      bits(64) preferred_exception_return, integer vect_offset_in)
assert HaveEL(target_el) && !ELUsingAArch32(target_el) && UInt(target_el) >= UInt(PSTATE.EL);
if Halted() then
    AArch64.TakeExceptionInDebugState(target_el, exception_in);
    return;
ExceptionRecord except = exception_in;
boolean sync_errors;
boolean iesb_req;
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);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        if except.exceptype != Exception\_SError then
            iesb_req = FALSE;
            sync_errors = FALSE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
    TMFailure cause;
    case except.exceptype of
        when Exception\_SoftwareBreakpoint cause = TMFailure\_DBG;
        when Exception\_Breakpoint cause = TMFailure\_DBG;
        when Exception\_Watchpoint cause = TMFailure\_DBG;
        when Exception\_SoftwareStep cause = TMFailure\_DBG;
        otherwise cause = TMFailure\_ERR;
    FailTransaction(cause, FALSE);

boolean brbe_source_allowed = FALSE;
bits(64) brbe_source_address = Zeros(64);
if IsFeatureImplemented(FEAT_BRBE) then
    brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
    brbe_source_address = preferred_exception_return;

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();

// If coming from AArch32 state, the top parts of the X[] registers might be set to zero
from_32 = UsingAArch32();
if from_32 then AArch64.MaybeZeroRegisterUppers();
if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
    ResetSVEState();
else
    MaybeZeroSVEUppers(target_el);

integer vect_offset = vect_offset_in;
if UInt(target_el) > UInt(PSTATE.EL) then
    boolean lower_32;
    if target_el == EL3 then
        if EL2Enabled() then
            lower_32 = ELUsingAArch32(EL2);
        else
            lower_32 = ELUsingAArch32(EL1);
    elseif IsInHost() && PSTATE.EL == EL0 && target_el == EL2 then
        lower_32 = ELUsingAArch32(EL0);

```

```

else
    lower_32 = ELUsingAArch32(target_el - 1);
    vect_offset = vect_offset + (if lower_32 then 0x600 else 0x400);

elsif PSTATE.SP == '1' then
    vect_offset = vect_offset + 0x200;
bits(64) spsr = GetPSRFromPSTATE(AArch64\_NonDebugState, 64);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if EffectiveHCR\_EL2\_NVx() IN {'x01', '111'} then
        spsr<3:2> = '10';

if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
    boolean zero_btype;
    // SPSR_ELx[].BTTYPE is only guaranteed valid for these exception types
    if except.excepttype 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\_ZEROBTTYPE);
    if zero_btype then spsr<11:10> = '00';

if (IsFeatureImplemented(FEAT_NV2) &&
    except.excepttype == Exception\_NV2DataAbort && target_el == EL3) then
    // External aborts are configured to be taken to EL3
    except.excepttype = Exception\_DataAbort;
if ! except.excepttype IN {Exception\_IRQ, Exception\_FIQ} then
    AArch64.ReportException(except, target_el);

if IsFeatureImplemented(FEAT_BRBE) then
    constant bits(64) brbe_target_address = VBAR\_EL[target_el]<63:11>:vect_offset<10:0>;
    BRBEException(except, brbe_source_allowed, brbe_source_address,
        brbe_target_address, target_el,
        except.trappedsyscallinst);

if IsFeatureImplemented(FEAT_GCS) then
    if PSTATE.EL == target_el then
        if GetCurrentEXLOCKEN() then
            PSTATE.EXLOCK = '1';
        else
            PSTATE.EXLOCK = '0';
    else
        PSTATE.EXLOCK = '0';

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR\_ELx[] = spsr;
ELR\_ELx[] = preferred_exception_return;

PSTATE.SS = '0';
if IsFeatureImplemented(FEAT_NMI) then
    PSTATE.ALLINT = NOT SCTLRL_ELx[].SPINTMASK;
PSTATE.<D,A,I,F> = '1111';
PSTATE.IL = '0';
if from_32 then // Coming from AArch32
    PSTATE.IT = '00000000';
    PSTATE.T = '0'; // PSTATE.J is RES0
if (IsFeatureImplemented(FEAT_PAN) && (PSTATE.EL == EL1 ||
    (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLRL_ELx[].SPAN == '0') then
    PSTATE.PAN = '1';
if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTTYPE = '00';
if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = SCTLRL_ELx[].DSSBS;

```

```

if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';
if IsFeatureImplemented(FEAT_EBEP) then PSTATE.PM = '1';
if IsFeatureImplemented(FEAT_SEBEP) then
    PSTATE.PPEND = '0';
    ShouldSetPPEND = FALSE;
constant boolean branch_conditional = FALSE;
BranchTo(VBAR_ELx[] <63:11>: vect_offset <10:0>, BranchType_EXCEPTION, branch_conditional);

CheckExceptionCatch(TRUE); // Check for debug event on exception entry

if sync_errors then
    SynchronizeErrors();
    iesb_req = TRUE;
    TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);

EndOfInstruction();

```

## Library pseudocode for aarch64/exceptions/traps/AArch64.AArch32SystemAccessTrap

```

// AArch64.AArch32SystemAccessTrap()
// =====
// Trapped AARCH32 System register access.

AArch64.AArch32SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

```



```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS,
// VMRS instructions, other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord except;

    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();

    bits(20) iss = Zeros(20);

    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
        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';

        if instr<20> == '1' && instr<15:12> == '1111' then // MRC, Rt==15
            iss<9:5> = '11111';
        elseif instr<20> == '0' && instr<15:12> == '1111' then // MCR, Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
    elseif except.exceptype IN {Exception\_CP14RRTTrap, Exception\_AdvSIMDFPAccessTrap,
                               Exception\_CP15RRTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMRS
        iss<19:16> = instr<7:4>; // opc1
        if instr<19:16> == '1111' then // Rt2==15
            iss<14:10> = bits(5) UNKNOWN;
        else
            iss<14:10> = LookUpRIndex(UInt(instr<19:16>), PSTATE.M)<4:0>;

        if instr<15:12> == '1111' then // Rt==15
            iss<9:5> = bits(5) UNKNOWN;
        else
            iss<9:5> = LookUpRIndex(UInt(instr<15:12>), PSTATE.M)<4:0>;
        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<9:5> = bits(5) UNKNOWN;
            iss<3>   = '1';
        iss<0> = instr<20>; // Direction

    except.syndrome.iss<24:20> = ConditionSyndrome();
    except.syndrome.iss<19:0> = iss;

```

```
return except;
```

### Library pseudocode for aarch64/exceptions/traps/AArch64.AdvSIMDFPAccessTrap

```
// AArch64.AdvSIMDFPAccessTrap()
// =====
// Trapped access to Advanced SIMD or FP registers due to CPACR[].

AArch64.AdvSIMDFPAccessTrap(bits(2) target_el)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0x0;

    route_to_el2 = (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);

    return;
```

### 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 SCTL_ELx.

AArch64.CheckCP15InstrCoarseTraps(integer CRn, integer nreg, integer CRm)
    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 MRC and MCR disabled by SCTL_EL1.TIDCP.
    if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && !IsInHost() &&
        !ELUsingAArch32(EL1) && SCTL_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTL_EL2.TIDCP.
        if (IsFeatureImplemented(FEAT_TIDCP1) && PSTATE.EL == EL0 && IsInHost() &&
            SCTL_EL2.TIDCP == '1' && trapped_encoding) then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);

    major = if nreg == 1 then CRn else CRm;
    // Check for MCR, MRC, MCRR, and MRRC disabled by HSTR_EL2<CRn/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 &&
            boolean IMPLEMENTATION_DEFINED "UNDEF unallocated CP15 access at EL0") then
            UNDEFINED;
        AArch64.AArch32SystemAccessTrap(EL2, 0x3);
```



## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDEnabled

```
// AArch64.CheckFPAdvSIMDEnabled()
// =====

AArch64.CheckFPAdvSIMDEnabled()
    AArch64.CheckFPEnabled\(\);
    // Check for illegal use of Advanced
    // SIMD in Streaming SVE Mode
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled\(\) then
        SMEAccessTrap\(SMEExceptionType\_Streaming, PSTATE.EL\);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPAdvSIMDTrap

```
// AArch64.CheckFPAdvSIMDTrap()
// =====
// Check against CPTR_EL2 and CPTR_EL3.

AArch64.CheckFPAdvSIMDTrap()
    if HaveEL\(EL3\) && CPTR_EL3.TFP == '1' && EL3SDDUndefPriority\(\) then
        UNDEFINED;

    if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled\(\) then
        // Check if access disabled in CPTR_EL2
        if ELIsInHost\(EL2\) then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = PSTATE.EL == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then AArch64.AdvSIMDFPAccessTrap\(EL2\);
        else
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap\(EL2\);

    if HaveEL\(EL3\) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef\(\) then
                UNDEFINED;
            else
                AArch64.AdvSIMDFPAccessTrap\(EL3\);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckFPEnabled

```
// AArch64.CheckFPEnabled()
// =====
// Check against CPACR[]

AArch64.CheckFPEnabled()
    if PSTATE.EL IN {EL0, EL1} && !IsInHost\(\) then
        // Check if access disabled in CPACR_EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = PSTATE.EL == EL0;
            when '11' disabled = FALSE;
        if disabled then AArch64.AdvSIMDFPAccessTrap\(EL1\);

    AArch64.CheckFPAdvSIMDTrap\(\); // Also check against CPTR_EL2 and CPTR_EL3
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForERetTrap

```
// AArch64.CheckForERetTrap()
// =====
// Check for trap on ERET, ERETAA, ERETAB instruction

AArch64.CheckForERetTrap(boolean eret_with_pac, boolean pac_uses_key_a)

    route_to_el2 = FALSE;
    // Non-secure EL1 execution of ERET, ERETAA, ERETAB when either HCR_EL2.NV or
    // HFGITR_EL2.ERET is set, is trapped to EL2
    route_to_el2 = (PSTATE.EL == EL1 && EL2Enabled() &&
        (EffectiveHCR_EL2_NVx() <0> == '1' ||
        (IsFeatureImplemented(FEAT_FGT) && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord except;
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = 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';
            if pac_uses_key_a then // ERETAA
                except.syndrome.iss<0> = '0';
            else // ERETAB
                except.syndrome.iss<0> = '1';
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSMCUnDefOrTrap

```
// AArch64.CheckForSMCUnDefOrTrap()
// =====
// Check for UNDEFINED or trap on SMC instruction

AArch64.CheckForSMCUnDefOrTrap(bits(16) imm)
    if PSTATE.EL == EL0 then UNDEFINED;
    if (!(PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1') &&
        HaveEL(EL3) && SCR_EL3.SMD == '1') then
        UNDEFINED;
    route_to_el2 = FALSE;
    if !HaveEL(EL3) then
        if (PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1' &&
            (EffectiveHCR_EL2_NVx() == 'xx1' ||
            (boolean IMPLEMENTATION_DEFINED "Trap SMC execution at EL1 to EL2"))) then
            route_to_el2 = TRUE;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;
        except = ExceptionSyndrome(Exception\_MonitorCall);
        except.syndrome.iss<15:0> = imm;
        except.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForSVCTrap

```
// AArch64.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch64.CheckForSVCTrap(bits(16) immediate)
    if IsFeatureImplemented(FEAT_FGT) then
        route_to_el2 = 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'));

        if route_to_el2 then
            except = ExceptionSyndrome(Exception\_SupervisorCall);
            except.syndrome.iss<15:0> = immediate;
            except.trappedsyscallinst = TRUE;
            constant bits(64) preferred_exception_return = ThisInstrAddr(64);
            vect_offset = 0x0;

            AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFXTrap

```
// AArch64.CheckForWFXTrap()
// =====
// Checks for a trap on a WFE, WFET, WFI or WFIT instruction.

(boolean, bits(2)) AArch64.CheckForWFXTrap(WFXType wfxttype)
    constant boolean is_wfe = wfxttype IN {WFXType\_WFE, WFXType\_WFET};
    bits(2) target_el;
    boolean trap = 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;

    if !trap && PSTATE.EL == EL0 then
        // Check for traps described by the OS which may be EL1 or EL2.
        trap = (if is_wfe then SCTLR_ELx[].nTWE else SCTLR_ELx[].nTWI) == '0';
        target_el = EL1;

    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;

    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;

    return (trap, target_el);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.CheckIllegalState

```
// AArch64.CheckIllegalState()
// =====
// Check PSTATE.IL bit and generate Illegal Execution state exception if set.

AArch64.CheckIllegalState()
    if PSTATE.IL == '1' then
        route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

        constant bits(64) preferred_exception_return = ThisInstrAddr(64);
        vect_offset = 0x0;

        except = 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);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.MonitorModeTrap

```
// AArch64.MonitorModeTrap()
// =====
// Trapped use of Monitor mode features in a Secure EL1 AArch32 mode

AArch64.MonitorModeTrap()
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_Uncategorized);

    if IsSecureEL2Enabled() then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
        AArch64.TakeException(EL3, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrap

```
// AArch64.SystemAccessTrap()
// =====
// Trapped access to AArch64 System register or system instruction.

AArch64.SystemAccessTrap(bits(2) target_el, integer ec)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.SystemAccessTrapSyndrome

```
// AArch64.SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch64 MSR/MRS instructions.

ExceptionRecord AArch64.SystemAccessTrapSyndrome(bits(32) instr_in, integer ec)
    ExceptionRecord except;
    bits(32) instr = instr_in;
    case ec of
        when 0x0 // Trapped access due to unknown reason.
            except = ExceptionSyndrome(ExceptionUncategorized);
        when 0x7 // Trapped access to SVE, Advance SIMD&FP System register.
            except = ExceptionSyndrome(ExceptionAdvSIMDFPAccessTrap);
            except.syndrome.iss<24:20> = ConditionSyndrome();
        when 0x14 // Trapped access to 128-bit System register or
            // 128-bit System instruction.
            except = ExceptionSyndrome(ExceptionSystemRegister128Trap);
            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(ExceptionSystemRegisterTrap);
            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(ExceptionSVEAccessTrap);
        when 0x1D // Trapped access to SME System register
            except = ExceptionSyndrome(ExceptionSMEAccessTrap);
        otherwise
            Unreachable();

    return except;
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.Undefined

```
// AArch64.Undefined()
// =====

AArch64.Undefined()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(ExceptionUncategorized);

    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);
```

## Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====
// Generate an exception for a trapped WFE, WFI, WFET or WFIT instruction.

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    ExceptionRecord except = ExceptionSyndrome(Exception_WFxTrap);
    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>;

    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);
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPAdvSIMDEnabled64

```
// CheckFPAdvSIMDEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPAdvSIMDEnabled64()
    AArch64.CheckFPAdvSIMDEnabled();
```

## Library pseudocode for aarch64/exceptions/traps/CheckFPEnabled64

```
// CheckFPEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPEnabled64()
    AArch64.CheckFPEnabled();
```

## Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
    boolean trap = FALSE;
    constant bits(25) iss = ZeroExtend('10', 25); // 0x2
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnALS == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnALS == '0';
            target_el = EL2;
    else
        target_el = EL1;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnALS == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/CheckST64BV0Enabled

```
// CheckST64BV0Enabled()
// =====
// Checks for trap on ST64BV0 instruction

CheckST64BV0Enabled()
    boolean trap = FALSE;
    constant bits(25) iss = ZeroExtend('1', 25); // 0x1
    bits(2) target_el;

    if (PSTATE.EL != EL3 && HaveEL(EL3) &&
        SCR_EL3.EnAS0 == '0' && EL3SDDUndefPriority()) then
        UNDEFINED;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnAS0 == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnAS0 == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnAS0 == '0';
        target_el = EL2;

    if !trap && PSTATE.EL != EL3 then
        trap = HaveEL(EL3) && SCR_EL3.EnAS0 == '0';
        target_el = EL3;

    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        else
            LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/CheckST64BVEnabled

```
// CheckST64BVEnabled()
// =====
// Checks for trap on ST64BV instruction

CheckST64BVEnabled()
    boolean trap = FALSE;
    constant bits(25) iss = Zeros(25);
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTLR_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTLR_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() &&
        ((PSTATE.EL == EL0 && !IsInHost()) || PSTATE.EL == EL1)) then
        trap = !IsHCRXEL2Enabled() || HCRX_EL2.EnASR == '0';
        target_el = EL2;

    if trap then LDST64BTrap(target_el, iss);
```

## Library pseudocode for aarch64/exceptions/traps/LDST64BTrap

```
// LDST64BTrap()
// =====
// Trapped access to LD64B, ST64B, ST64BV and ST64BV0 instructions

LDST64BTrap(bits(2) target_el, bits(25) iss)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception\_LDST64BTrap);
    except.syndrome.iss = iss;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);

    return;
```

## 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.

(boolean, integer) WFETrapDelay(bits(2) target_el)
    boolean delay_enabled;
    integer delay;
    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);
        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);
    return (delay_enabled, delay);
```



## Library pseudocode for aarch64/exceptions/traps/WaitForEventUntilDelay

```
// WaitForEventUntilDelay()
// =====
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.

boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```



```

// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value and updates the virtual address for Abort and Watchpoint
// exceptions taken to an Exception level using AArch64.

IssType AArch64.FaultSyndrome(Exception exceptype, FaultRecord fault, boolean pavalid)
    assert fault.statuscode != Fault_None;

    IssType isstype;
    isstype.iss = Zeros(25);
    isstype.iss2 = Zeros(24);

    constant boolean d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort,
                                             Exception_Watchpoint, Exception_NV2Watchpoint};

    if IsFeatureImplemented(FEAT_RAS) && fault.statuscode == Fault_SyncExternal then
        constant ErrorState errstate = PEErrState(fault);
        isstype.iss<12:11> = AArch64.EncodeSyncErrorSyndrome(errstate); // SET

    if d_side then
        if fault.accessdesc.acctype == AccessType_GCS then
            isstype.iss2<8> = '1';
        if exceptype IN {Exception_Watchpoint, Exception_NV2Watchpoint} then
            isstype.iss<23:0> = WatchpointRelatedSyndrome(fault);
        if IsFeatureImplemented(FEAT_LS64) && fault.accessdesc.ls64 then
            if (fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
                (isstype.iss2, isstype.iss<24:14>) = LS64InstructionSyndrome();
            elsif (IsSecondStage(fault) && !fault.s2fslwalk &&
                    (!IsExternalSyncAbort(fault) ||
                     (!IsFeatureImplemented(FEAT_RAS) && fault.accessdesc.acctype == AccessType_TTW &&
                      boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk"))) then
                isstype.iss<24:14> = LSInstructionSyndrome();

        if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType_NV2 then
            isstype.iss<13> = '1'; // Fault is generated by use of VNCR_EL2

        if (IsFeatureImplemented(FEAT_LS64) &&
            fault.statuscode IN {Fault_AccessFlag, Fault_Translation, Fault_Permission}) then
            isstype.iss<12:11> = GetLoadStoreType();

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<8> = '1';

        if fault.accessdesc.acctype IN {AccessType_DC, AccessType_IC, AccessType_AT} then
            isstype.iss<6> = '1';
        elsif fault.statuscode IN {Fault_HWUpdateAccessFlag, Fault_Exclusive} then
            isstype.iss<6> = bit UNKNOWN;
        elsif fault.accessdesc.atomicop && IsExternalAbort(fault) then
            isstype.iss<6> = bit UNKNOWN;
        else
            isstype.iss<6> = if fault.write 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';
                isstype.iss2<7> = if fault.assuredonly then '1' else '0';
                isstype.iss2<9> = if fault.tagaccess then '1' else '0';
                isstype.iss2<10> = if fault.sltagnotdata then '1' else '0';

    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';
        isstype.iss2<11> = if fault.hdbssf then '1' else '0';

    if IsExternalAbort(fault) then isstype.iss<9> = fault.extflag;
    isstype.iss<7> = if fault.s2fslwalk then '1' else '0';

```

```

issstype.iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

return issstype;

```

### Library pseudocode for aarch64/functions/aborts/EncodeGPCSC

```

// EncodeGPCSC()
// =====
// Function that gives the GPCSC code for types of GPT Fault

bits(6) EncodeGPCSC(GPCFRecord gpcf)
    assert gpcf.level IN {0,1};

    case gpcf.gpf 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>;

```

### 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.

(bits(24), bits(11)) LS64InstructionSyndrome();

```

### 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.

boolean WatchpointFARNotPrecise(FaultRecord fault);

```

### Library pseudocode for aarch64/functions/apas/AArch64.APAS

```

// AArch64.APAS()
// =====
// Decode Xt and perform an APAS operation for the decoded record.

AArch64.APAS(bits(64) Xt)
    APASRecord apas;
    constant bit nse2 = '0';
    apas.paspace = DecodePASpace(nse2, Xt<62>, Xt<63>);

    apas.pa = Xt<55:6> : '000000';
    apas.target_attributes = Xt<2:0>;

    if AArch64.LocationSupportsAPAS(apas) then
        APAS\_OP(apas);

```

### Library pseudocode for aarch64/functions/apas/AArch64.LocationSupportsAPAS

```

// AArch64.LocationSupportsAPAS()
// =====
// Returns TRUE if the given memory location supports the APAS instruction.

boolean AArch64.LocationSupportsAPAS(APASRecord apas);

```

## Library pseudocode for aarch64/functions/apas/APASRecord

```
// APASRecord
// =====
// Details related to an APAS operation.

type APASRecord is (
    bits(56) pa,
    PASpace paspace,
    bits(3) target_attributes
)
```

## 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.

APAS_OP(APASRecord apas)
    IMPLEMENTATION_DEFINED;
```



```

// AArch64.AT()
// =====
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage, bits(2) el_in, ATAccess ataccess)
    bits(2) el = el_in;
    constant bits(2) effective_nse_ns = EffectiveSCR\_EL3\_NSE() : EffectiveSCR\_EL3\_NS();
    if (IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3 &&
        effective_nse_ns == '10' && el != EL3) then
        UNDEFINED;
    // 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;

    constant SecurityState ss = SecurityStateAtEL(el);

    accdesc = CreateAccDescAT(ss, el, ataccess);
    aligned = TRUE;

    FaultRecord fault = NoFault(accdesc, address);
    Regime regime;
    if stage == TranslationStage\_12 then
        regime = Regime\_EL10;
    else
        regime = TranslationRegime(el);

    AddressDescriptor addrdesc;
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime\_EL2 || TBCR.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);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, address, aligned, accdesc);

    if stage == TranslationStage\_12 && fault.statuscode == Fault\_None then
        constant boolean slaarch64 = TRUE;
        if ELUsingAArch32(EL1) && regime == Regime\_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address, 64);
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, aligned, accdesc);
        elsif regime == Regime\_EL10 && EL2Enabled() then
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, slaarch64, aligned, accdesc);

    is_ATS1Ex = stage != TranslationStage\_12;
    if fault.statuscode != Fault\_None then
        addrdesc = CreateFaultyAddressDescriptor(address, 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 && (
                ReportAsGPCEException(fault) ||
                (EL2Enabled() && HCR_EL2.GPF == '1' && PSTATE.EL == EL1 && el IN {EL1, EL0} &&
                    is_ATS1Ex)
            ))) then
            if IsFeatureImplemented(FEAT_D128) then
                PAR_EL1 = bits(128) UNKNOWN;
            else
                PAR_EL1<63:0> = bits(64) UNKNOWN;
            AArch64.Abort(addrdesc.fault);

    AArch64.EncodePAR(regime, is_ATS1Ex, addrdesc);
    return;

```

## Library pseudocode for aarch64/functions/at/AArch64.EncodePAR

```
// AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, boolean is_ATS1Ex, AddressDescriptor addrdesc)
    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';
    else
        PAR_EL1<63:0> = Zeros(64);

    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';

                elseif SecurityStateForRegime(regime) == SS\_Secure then
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = if paspace == PAS\_Secure then '0' else '1';

                elseif SecurityStateForRegime(regime) == SS\_Realm then
                    if regime == Regime\_EL10 && is_ATS1Ex then
                        PAR_EL1.NSE = bit UNKNOWN;
                        PAR_EL1.NS = bit UNKNOWN;
                    else
                        PAR_EL1.NSE = bit UNKNOWN;
                        PAR_EL1.NS = if paspace == PAS\_Realm then '0' else '1';

                else
                    PAR_EL1.NSE = bit UNKNOWN;
                    PAR_EL1.NS = bit UNKNOWN;
            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 = bit UNKNOWN;
        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>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "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> = bits(16) IMPLEMENTATION_DEFINED "Faulting PAR";
    return;
```



## Library pseudocode for aarch64/functions/at/AArch64.PARFaultStatus

```
// AArch64.PARFaultStatus()
// =====
// Fault status field decoding of 64-bit PAR.

bits(6) AArch64.PARFaultStatus(FaultRecord fault)
    bits(6) fst;

    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);
    return fst;
```

## 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.

boolean AArch64.isPARFormatD128(Regime regime, boolean is_ATS1Ex)
    boolean isPARFormatD128;
    // 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 = TCR2_EL2.D128 == '1';
            when Regime\_EL10
                if is_ATS1Ex || !EL2Enabled() || HCR_EL2.<VM,DC> == '00' then
                    isPARFormatD128 = TCR2_EL1.D128 == '1';
                else
                    isPARFormatD128 = VTCR_EL2.D128 == '1';

    return isPARFormatD128;
```

## Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_D128

```
// GetPAR_EL1_D128()
// =====
// Query the PAR_EL1.D128 field

bit GetPAR_EL1_D128()
    return if IsFeatureImplemented(FEAT_D128) then PAR_EL1.D128 else '0';
```

## Library pseudocode for aarch64/functions/at/GetPAR\_EL1\_F

```
// GetPAR_EL1_F()
// =====
// Query the PAR_EL1.F field.

bit GetPAR_EL1_F()
    bit F;

    F = PAR_EL1.F;
    return F;
```

## Library pseudocode for aarch64/functions/barrierop/MemBarrierOp

```
// MemBarrierOp
// =====
// Memory barrier instruction types.

enumeration MemBarrierOp    {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.

boolean BFXPreferred(bit sf, bit uns, bits(6) imms, bits(6) immr)

    // must not match UBFIZ/SBFIX alias
    if UInt(imms) < UInt(immr) then
        return FALSE;

    // must not match LSR/ASR/LSL alias (imms == 31 or 63)
    if imms == sf:'11111' then
        return FALSE;

    // 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;
        // must not match 64-bit SXT[BHW]
        if sf:uns == '10' && imms IN {'000111', '001111', '011111'} then
            return FALSE;

    // must be UBFX/SBFX alias
    return TRUE;
```



```

// AltDecodeBitMasks()
// =====
// Alternative but logically equivalent implementation of DecodeBitMasks() that
// uses simpler primitives to compute tmask and wmask.

(bits(M), bits(M)) AltDecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                     boolean immediate, integer M)

    bits(64) tmask, wmask;
    bits(6) tmask_and, wmask_and;
    bits(6) tmask_or, wmask_or;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    constant integer len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // 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;

    s = UInt(imms AND levels);
    r = UInt(immr AND levels);
    diff = 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(Replicate(tmask_and<0>, 1) : Ones(1), 32))
        OR Replicate(Zeros(1) : Replicate(tmask_or<0>, 1), 32));
    // optimization of first step:
    // tmask = Replicate(tmask_and<0> : '1', 32);
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<1>, 2) : Ones(2), 16))
        OR Replicate(Zeros(2) : Replicate(tmask_or<1>, 2), 16));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<2>, 4) : Ones(4), 8))
        OR Replicate(Zeros(4) : Replicate(tmask_or<2>, 4), 8));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<3>, 8) : Ones(8), 4))
        OR Replicate(Zeros(8) : Replicate(tmask_or<3>, 8), 4));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<4>, 16) : Ones(16), 2))
        OR Replicate(Zeros(16) : Replicate(tmask_or<4>, 16), 2));
    tmask = ((tmask
        AND Replicate(Replicate(tmask_and<5>, 32) : Ones(32), 1))
        OR Replicate(Zeros(32) : Replicate(tmask_or<5>, 32), 1));

    // Compute "wraparound mask"
    wmask_and = immr OR NOT(levels);
    wmask_or  = immr AND levels;

    wmask = Zeros(64);
    wmask = ((wmask
        AND Replicate(Ones(1) : Replicate(wmask_and<0>, 1), 32))
        OR Replicate(Replicate(wmask_or<0>, 1) : Zeros(1), 32));
    // optimization of first step:
    // wmask = Replicate(wmask_or<0> : '0', 32);
    wmask = ((wmask
        AND Replicate(Ones(2) : Replicate(wmask_and<1>, 2), 16))
        OR Replicate(Replicate(wmask_or<1>, 2) : Zeros(2), 16));
    wmask = ((wmask

```

```

        AND Replicate(Ones(4) : Replicate(wmask_and<2>, 4), 8))
        OR Replicate(Replicate(wmask_or<2>, 4) : Zeros(4), 8));
wmask = ((wmask
        AND Replicate(Ones(8) : Replicate(wmask_and<3>, 8), 4))
        OR Replicate(Replicate(wmask_or<3>, 8) : Zeros(8), 4));
wmask = ((wmask
        AND Replicate(Ones(16) : Replicate(wmask_and<4>, 16), 2))
        OR Replicate(Replicate(wmask_or<4>, 16) : Zeros(16), 2));
wmask = ((wmask
        AND Replicate(Ones(32) : Replicate(wmask_and<5>, 32), 1))
        OR Replicate(Replicate(wmask_or<5>, 32) : Zeros(32), 1));

if diff<6> != '0' then // borrow from s - r
    wmask = wmask AND tmask;
else
    wmask = wmask OR tmask;

return (wmask<M-1:0>, tmask<M-1:0>);

```

## Library pseudocode for aarch64/functions/bitmasks/DecodeBitMasks

```

// DecodeBitMasks()
// =====
// Decode AArch64 bitfield and logical immediate masks which use a similar encoding structure

(bits(M), bits(M)) DecodeBitMasks(bit immN, bits(6) imms, bits(6) immr,
                                   boolean immediate, integer M)

    bits(M) tmask, wmask;
    bits(6) levels;

    // Compute log2 of element size
    // 2^len must be in range [2, M]
    constant integer len = HighestSetBit(immN:NOT(imms));
    if len < 1 then UNDEFINED;
    assert M >= (1 << len);

    // Determine s, r and s - r parameters
    levels = ZeroExtend(Ones(len), 6);

    // 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;

    constant integer s = UInt(imms AND levels);
    constant integer r = UInt(immr AND levels);
    constant integer diff = s - r;    // 6-bit subtract with borrow

    constant integer esize = 1 << len;
    constant integer d = UInt(diff<len-1:0>);
    welem = ZeroExtend(Ones(s + 1), esize);
    telem = ZeroExtend(Ones(d + 1), esize);
    wmask = Replicate(ROR(welem, r), M DIV esize);
    tmask = Replicate(telem, M DIV esize);
    return (wmask, tmask);

```

## Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====
// Write Zero to data memory.

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
AccessDescriptor accdesc = 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 = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

// 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);

constant boolean aligned = TRUE;
AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

if IsFault(memaddrdesc) then
    if !IsDebugException(memaddrdesc.fault) then
        memaddrdesc.fault.vaddress = regval;
        AArch64.Abort(memaddrdesc.fault);

if IsFeatureImplemented(FEAT_TME) then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattr) then
        FailTransaction(TMFailure\_IMP, FALSE);

for i = 0 to size-1
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(vaddress);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            if (boolean IMPLEMENTATION_DEFINED
                "DC_ZVA tag fault reported with lowest faulting address") then
                AArch64.TagCheckFault(vaddress, accdesc);
            else
                AArch64.TagCheckFault(regval, accdesc);
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros(8));
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;
```

## Library pseudocode for aarch64/functions/cache/AArch64.WriteTagMem

```
// AArch64.WriteTagMem()
// =====
// Write to tag memory.

AArch64.WriteTagMem(bits(64) regval, bits(64) vaddress, AccessDescriptor accdesc_in, integer size)
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AccessDescriptor accdesc = accdesc_in;

    constant integer count = size >> LOG2\_TAG\_GRANULE;
    constant bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    constant boolean aligned = IsAligned(vaddress, TAG\_GRANULE);
    assert aligned;

    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        if !IsDebugException(memaddrdesc.fault) then
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

    if !accdesc.tagaccess || memaddrdesc.memattrs.tags != MemTag\_AllocationTagged then
        return;

    for i = 0 to count-1
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, tag);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + TAG\_GRANULE;

    return;
```

## Library pseudocode for aarch64/functions/compareop/CompareOp

```
// CompareOp
// =====
// Vector compare instruction types.

enumeration CompareOp    {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                          CompareOp_LE, CompareOp_LT};
```

## Library pseudocode for aarch64/functions/countop/CountOp

```
// CountOp
// =====
// Bit counting instruction types.

enumeration CountOp      {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.

bit EffectiveCPTA(bits(2) el)
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';

    if Halted\(\) then
        return '0';

    bits(1) cpta;
    constant 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';
        when Regime\_EL20
            if IsSCTLR2EL2Enabled\(\) then
                cpta = if el == ELO then SCTLR2_EL2.CPTA0 else SCTLR2_EL2.CPTA;
            else
                cpta = '0';
        when Regime\_EL10
            if IsSCTLR2EL1Enabled\(\) then
                cpta = if el == ELO then SCTLR2_EL1.CPTA0 else SCTLR2_EL1.CPTA;
            else
                cpta = '0';
        otherwise
            Unreachable();

    return cpta;
```



## Library pseudocode for aarch64/functions/cpa/EffectiveCPTM

```
// EffectiveCPTM()
// =====
// Returns the CPTM bit applied to Checked Pointer Arithmetic for Multiplication in the given EL.

bit EffectiveCPTM(bits(2) el)
    if !IsFeatureImplemented(FEAT_CPA2) then
        return '0';

    if EffectiveCPTA(el) == '0' then
        return '0';

    if Halted() then
        return '0';

    bits(1) cptm;
    constant 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';
        when Regime\_EL20
            if IsSCTLR2EL2Enabled() then
                cptm = if el == ELO then SCTLR2_EL2.CPTM0 else SCTLR2_EL2.CPTM;
            else
                cptm = '0';
        when Regime\_EL10
            if IsSCTLR2EL1Enabled() then
                cptm = if el == ELO then SCTLR2_EL1.CPTM0 else SCTLR2_EL1.CPTM;
            else
                cptm = '0';
        otherwise
            Unreachable();

    return cptm;
```

## Library pseudocode for aarch64/functions/cpa/PointerAddCheck

```
// PointerAddCheck()
// =====
// Apply Checked Pointer Arithmetic for addition.

bits(64) PointerAddCheck(bits(64) result, bits(64) base)
    return PointerCheckAtEL(PSTATE.EL, result, base, FALSE);
```

## Library pseudocode for aarch64/functions/cpa/PointerAddCheckAtEL

```
// PointerAddCheckAtEL()
// =====
// Apply Checked Pointer Arithmetic for addition at the specified EL.

bits(64) PointerAddCheckAtEL(bits(2) el, bits(64) result, bits(64) base)
    return PointerCheckAtEL(el, result, base, FALSE);
```

## Library pseudocode for aarch64/functions/cpa/PointerCheckAtEL

```
// PointerCheckAtEL()
// =====
// Apply Checked Pointer Arithmetic at the specified EL.

bits(64) PointerCheckAtEL(bits(2) el, bits(64) result, bits(64) base, boolean cptm_detected)
    bits(64) rv = result;

    constant boolean previous_detection = (base<55> != base<54>);
    constant boolean cpta_detected = (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>);

    return rv;
```

## Library pseudocode for aarch64/functions/cpa/PointerMultiplyAddCheck

```
// PointerMultiplyAddCheck()
// =====
// Apply Checked Pointer Arithmetic for multiplication.

bits(64) PointerMultiplyAddCheck(bits(64) result, bits(64) base, boolean cptm_detected)
    return PointerCheckAtEL(PSTATE.EL, result, base, cptm_detected);
```

## Library pseudocode for aarch64/functions/d128/IsD128Enabled

```
// IsD128Enabled()
// =====
// Returns true if 128-bit page descriptor is enabled

boolean IsD128Enabled(bits(2) el)
    boolean d128enabled;
    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';
            when EL1
                d128enabled = IsTCR2EL1Enabled() && TCR2_EL1.D128 == '1';
            when EL2
                d128enabled = IsTCR2EL2Enabled() && IsInHost() && TCR2_EL2.D128 == '1';
            when EL3
                d128enabled = TCR_EL3.D128 == '1';
        else
            d128enabled = FALSE;

    return d128enabled;
```



```

// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
CacheOpScope opscope = opscope_in;
CacheRecord cache;

cache.acctype = AccessType\_DC;
cache.cachetype = cachetype;
cache.cacheop = cacheop;
cache.opscope = opscope;

if opscope == CacheOpScope\_SetWay then
    ss = SecurityStateAtEL(PSTATE.EL);
    cache.cpas = CPASAtSecurityState(ss);
    cache.shareability = Shareability\_NSH;
    (cache.setnum, cache.waynum, cache.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;

    CACHE\_OP(cache);
    return;

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;
else
    cache.is_vmid_valid = FALSE;

if PSTATE.EL == EL0 then
    cache.is_asid_valid = TRUE;
    cache.asid = ASID[];
else
    cache.is_asid_valid = FALSE;

if (opscope == CacheOpScope\_PoPS &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not support PoPS") then
    opscope = CacheOpScope\_PoC;
if (opscope == CacheOpScope\_PoDP &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not support PoDP") then
    opscope = CacheOpScope\_PoP;
if (opscope == CacheOpScope\_PoP &&
    boolean IMPLEMENTATION_DEFINED "Memory system does not support PoP") then
    opscope = CacheOpScope\_PoC;
vaddress = regval;

integer size = 0; // by default no watchpoint address
if cacheop == CacheOp\_Invalidate then
    size = DataCacheWatchpointSize();
    vaddress = Align(regval, size);

if DCInstNeedsTranslation(opscope) then
    cache.vaddress = vaddress;
    constant boolean aligned = TRUE;
    constant AccessDescriptor accdesc = CreateAccDescDC(cache);
    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc,
                                                                aligned, size);

    if IsFault(memaddrdesc) then
        memaddrdesc.fault.vaddress = regval;
        AArch64.Abort(memaddrdesc.fault);

    cache.translated = TRUE;
    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;
elsif opscope == CacheOpScope\_PoE then
    cache.translated = TRUE;
    cache.shareability = Shareability\_OSH;
    cache.paddress.address = regval<55:0>;
    constant bit nse2 = 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();

    if boolean IMPLEMENTATION_DEFINED "Apply granule protection check on DC to PoE" then
        AddressDescriptor memaddrdesc;
        constant AccessDescriptor accdesc = CreateAccDescDC(cache);
        memaddrdesc.paddress = cache.paddress;
        memaddrdesc.fault.gpcf = GranuleProtectionCheck(memaddrdesc, accdesc);

        if memaddrdesc.fault.gpcf.gpcf != GPCF\_None then
            memaddrdesc.fault.statuscode = Fault\_GPCFOnOutput;
            memaddrdesc.fault.paddress = memaddrdesc.paddress;
            memaddrdesc.fault.vaddress = bits(64) UNKNOWN;
            AArch64.Abort(memaddrdesc.fault);
    elsif opscope == CacheOpScope\_PoPA then
        cache.translated = TRUE;
        cache.shareability = Shareability\_OSH;
        cache.paddress.address = regval<55:0>;
        constant bit nse2 = 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);
    else
        cache.vaddress = vaddress;
        cache.translated = FALSE;
        cache.shareability = Shareability UNKNOWN;
        cache.paddress = FullAddress UNKNOWN;

    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        HCR_EL2.<DC,VM> != '00') then
        cache.cacheop = CacheOp\_CleanInvalidate;

    // If Secure state is not implemented, but RME is, the instruction acts as a NOP
    if cache.translated && cache.cpas == CPAS\_Secure && !HaveSecureState() then
        return;

    CACHE\_OP(cache);
    return;

```

## Library pseudocode for aarch64/functions/dc/AArch64.MemZero

```
// AArch64.MemZero()
// =====

AArch64.MemZero(bits(64) regval, CacheType cachetype)
    constant integer size = 4*(2^(UInt(DCZID_ELO.BS)));
    assert size <= MAX\_ZERO\_BLOCK\_SIZE;
    if IsFeatureImplemented(FEAT_MTE2) then
        assert size >= TAG\_GRANULE;

    constant bits(64) vaddress = Align(regval, size);

    constant AccessDescriptor accdesc = CreateAccDescDCZero(cachetype);

    if cachetype IN {CacheType\_Tag, CacheType\_Data\_Tag} then
        AArch64.WriteTagMem(regval, vaddress, accdesc, size);

    if cachetype IN {CacheType\_Data, CacheType\_Data\_Tag} then
        AArch64.DataMemZero(regval, vaddress, accdesc, size);
    return;
```

## Library pseudocode for aarch64/functions/dc/MemZero

```
// MemZero block size
// =====

constant integer MAX_ZERO_BLOCK_SIZE = 2048;
```

## Library pseudocode for aarch64/functions/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;

    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction\(TMFailure\_ERR, FALSE\);

    if IsFeatureImplemented(FEAT_IESB) then
        sync_errors = SCTLR_ELx[].IESB == '1';
        if IsFeatureImplemented(FEAT_DoubleFault) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors\(\);
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts\(iesb\_req\);

    boolean brbe_source_allowed = FALSE;
    bits(64) brbe_source_address = Zeros(64);
    if IsFeatureImplemented(FEAT_BRBE) then
        brbe_source_allowed = BranchRecordAllowed(PSTATE.EL);
        brbe_source_address = PC64;

    if !IsFeatureImplemented(FEAT_ExS) || SCTLR_ELx[].EOS == '1' then
        SynchronizeContext\(\);

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    constant bits(2) source_el = PSTATE.EL;
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(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> = bits(32) UNKNOWN;
        new_pc<1:0> = bits(2) UNKNOWN;
    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
    else // Return to AArch64
        // ELR_ELx[63:56] might include a tag
        new_pc = AArch64.BranchAddr(new_pc, PSTATE.EL);

    if IsFeatureImplemented(FEAT_BRBE) then
        BRBEEExceptionReturn(new_pc, source_el,
            brbe_source_allowed, brbe_source_address);

    if UsingAArch32() then
        if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then ResetSVEState();

        // 32 most significant bits are ignored.
        constant boolean branch_conditional = FALSE;
        BranchTo(new_pc<31:0>, BranchType\_ERET, branch_conditional);
    else
        BranchToAddr(new_pc, BranchType\_ERET);

    CheckExceptionCatch(FALSE); // Check for debug event on exception return
```

## 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 an memory abort.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size, AccessDescriptor accdesc)
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    if !AArch64.IsExclusiveVA(address, ProcessorID(), size) then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.paddress, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed && memaddrdesc.memattrs.shareability != Shareability\_NSH then
        passed = IsExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    return passed;
```

## 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.

boolean AArch64.IsExclusiveVA(bits(64) address, integer processorid, integer size);
```

## 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.

AArch64.MarkExclusiveVA(bits(64) address, integer processorid, integer size);
```



## 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.

AArch64.SetExclusiveMonitors(bits(64) address, integer size)
    constant boolean acqrel = FALSE;
    constant boolean privileged = PSTATE.EL != EL0;
    constant boolean tagchecked = FALSE;
    constant AccessDescriptor accdesc = CreateAccDescExLDST(MemOp\_LOAD, acqrel,
                                                    tagchecked, privileged);

    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.address, ProcessorID(), size);

    AArch64.MarkExclusiveVA(address, ProcessorID(), size);
```

## Library pseudocode for aarch64/functions/extendreg/DecodeRegExtend

```
// DecodeRegExtend()
// =====
// Decode a register extension option

ExtendType DecodeRegExtend(bits(3) op)
    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;
```

## Library pseudocode for aarch64/functions/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift, integer N)
    assert shift >= 0 && shift <= 4;
    constant bits(N) val = X[reg, N];
    boolean unsigned;
    ESize len;

    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\_SCTX 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\_UCTX unsigned = TRUE; len = 64;

    // Sign or zero extend bottom LEN bits of register and shift left by SHIFT
    constant nbits = Min(len, N);
    constant bits(N) extval = Extend(val<nbits-1:0>, N, unsigned);
    return LSL(extval, shift);
```

## Library pseudocode for aarch64/functions/extendreg/ExtendType

```
// ExtendType
// =====
// AArch64 register extend and shift.

enumeration ExtendType {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.

enumeration FPConvOp {FPConvOp\_CVT\_FtoI, FPConvOp\_CVT\_ItoF,
                     FPConvOp\_MOV\_FtoI, FPConvOp\_MOV\_ItoF,
                     FPConvOp\_CVT\_FtoI\_JS
};
```

## Library pseudocode for aarch64/functions/fpmaxminop/FPMaxMinOp

```
// FPMaxMinOp
// =====
// Floating-point min/max instruction types.

enumeration FPMaxMinOp {FPMaxMinOp\_MAX, FPMaxMinOp\_MIN,
                       FPMaxMinOp\_MAXNUM, FPMaxMinOp\_MINNUM};
```

## Library pseudocode for aarch64/functions/fpmr/CheckFPMREnabled

```
// CheckFPMREnabled()
// =====
// Check for Undefined instruction exception on indirect FPMR accesses.

CheckFPMREnabled()
    assert IsFeatureImplemented(FEAT_FPMR);

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            if SCTLRL_EL1.EnFPM == '0' then UNDEFINED;
        else
            if SCTLRL_EL2.EnFPM == '0' then UNDEFINED;

    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        if !IsHCRXEL2Enabled() || HCRX_EL2.EnFPM == '0' then UNDEFINED;

    if PSTATE.EL != EL3 && HaveEL(EL3) then
        if SCR_EL3.EnFPM == '0' then UNDEFINED;
```

## Library pseudocode for aarch64/functions/fpscale/FPScale

```
// FPScale()
// =====

bits(N) FPScale(bits(N) op, integer scale, FPCR_Type fpcr)
    assert N IN {16, 32, 64};
    bits(N) result;
    (fptype, sign, value) = FPUnpack(op, fpcr);

    if fptype == FPTYPE_SNaN || fptype == FPTYPE_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPTYPE_Infinity then
        result = FPInfinity(sign, N);
    else
        result = FPRound(value * (2.0^scale), fpcr, N);
        FPProcessDenorm(fptype, N, fpcr);
    return result;
```

## Library pseudocode for aarch64/functions/fpunaryop/FPUnaryOp

```
// FPUnaryOp
// =====
// Floating-point unary instruction types.

enumeration FPUnaryOp    {FPUnaryOp_ABS, FPUnaryOp_MOV,
                          FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRSqrtStepFused

```
// FPRSqrtStepFused()
// =====

bits(N) FPRSqrtStepFused(bits(N) op1_in, bits(N) op2, FPCR\_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR\_Type fpcr = fpcr_in;
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    op1 = FPNeg(op1, fpcr);
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPOnePointFive('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add and halve
            result_value = (3.0 + (value1 * value2)) / 2.0;
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

## Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2, FPCR\_Type fpcr_in)
    assert N IN {16, 32, 64};
    FPCR\_Type fpcr = fpcr_in;
    bits(N) op1 = op1_in;
    bits(N) result;
    boolean done;
    op1 = FPNeg(op1, fpcr);

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp;                                // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11';                             // Flush denormal input and output to zero
    if altfp then fpcr.RMode = '00';                                // Use RNE rounding mode

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    constant FPRounding rounding = FPRoundingMode(fpcr);

    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo('0', N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        else
            // Fully fused multiply-add
            result_value = 2.0 + (value1 * value2);
            if result_value == 0.0 then
                // Sign of exact zero result depends on rounding mode
                sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    return result;
```

## Library pseudocode for aarch64/functions/gcs/AddGCSExRecord

```
// AddGCSExRecord()
// =====
// Generates and then writes an exception record to the
// current Guarded control stack.

AddGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = lr;
    Mem[ptr-16, 8, accdesc] = spsr;
    Mem[ptr-24, 8, accdesc] = elr;
    Mem[ptr-32, 8, accdesc] = Zeros(60):'1001';

    // Decrement the pointer value
    ptr = ptr - 32;

    SetCurrentGCSPointer(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/AddGCSRecord

```
// AddGCSRecord()
// =====
// Generates and then writes a record to the current Guarded
// control stack.

AddGCSRecord(bits(64) vaddress)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_STORE, privileged);

    ptr = GetCurrentGCSPointer();

    // Store the record
    Mem[ptr-8, 8, accdesc] = vaddress;

    // Decrement the pointer value
    ptr = ptr - 8;

    SetCurrentGCSPointer(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/CheckGCSExRecord

```
// CheckGCSExRecord()
// =====
// Validates the provided values against the top entry of the
// current Guarded control stack.

CheckGCSExRecord(bits(64) elr, bits(64) spsr, bits(64) lr, GCSInstruction gcsinst_type)
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_LOAD, privileged);
    ptr = GetCurrentGCSPtr();

    // Check the lowest doubleword is correctly formatted
    constant bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(gcsinst_type);

    // Check the ELR matches the recorded value
    constant bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    if recorded_elr != elr then
        GCSDataCheckException(gcsinst_type);

    // Check the SPSR matches the recorded value
    constant bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    if recorded_spsr != spsr then
        GCSDataCheckException(gcsinst_type);

    // Check the LR matches the recorded value
    constant bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];
    if recorded_lr != lr then
        GCSDataCheckException(gcsinst_type);

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPtr(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/CheckGCSSTREnabled

```
// CheckGCSSTREnabled()
// =====
// Trap GCSSTR or GCSSTTR instruction if trapping is enabled.

CheckGCSSTREnabled()
    case PSTATE.EL of
        when EL0
            if GCSCRE0_EL1.STREn == '0' then
                if EL2Enabled() && HCR_EL2.TGE == '1' then
                    GCSSTRTrapException(EL2);
                else
                    GCSSTRTrapException(EL1);
            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);
            when EL2
                if GCSCR_EL2.STREn == '0' then
                    GCSSTRTrapException(EL2);
            when EL3
                if GCSCR_EL3.STREn == '0' then
                    GCSSTRTrapException(EL3);
    return;
```

## Library pseudocode for aarch64/functions/gcs/EXLOCKException

```
// EXLOCKException()
// =====
// Handle an EXLOCK exception condition.

EXLOCKException()
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;

    except = 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);
```

## Library pseudocode for aarch64/functions/gcs/GCSDataCheckException

```
// GCSDataCheckException()
// =====
// Handle a Guarded Control Stack data check fault condition.

GCSDataCheckException(GCSInstruction gcsinst_type)
    bits(2) target_el;
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    boolean rn_unknown = FALSE;
    boolean is_ret = FALSE;
    boolean is_reta = FALSE;

    if PSTATE.EL == EL0 then
        target_el = if (EL2Enabled() && HCR_EL2.TGE == '1') then EL2 else EL1;
    else
        target_el = PSTATE.EL;
    except = 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';
    if rn_unknown == TRUE then
        except.syndrome.iss<9:5> = bits(5) UNKNOWN;
    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>;
    except.syndrome.iss<24:10> = Zeros(15);
    except.vaddress = bits(64) UNKNOWN;
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```



## Library pseudocode for aarch64/functions/gcs/GCSEnabled

```
// GCSEnabled()
// =====
// Returns TRUE if the Guarded control stack is enabled at
// the provided Exception level.

boolean GCSEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;

    if HaveEL(EL3) && el != EL3 && SCR_EL3.GCSEn == '0' then
        return FALSE;

    if (el IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.GCSEn == '0')) then
        return FALSE;

    return GCSPCRSelected(el);
```

## Library pseudocode for aarch64/functions/gcs/GCSInstruction

```
// GCSInstruction
// =====

enumeration GCSInstruction {
    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.

boolean GCSPCREnabled(bits(2) el)
    return GCSPCRSelected(el) && GCSEnabled(el);
```

## Library pseudocode for aarch64/functions/gcs/GCSPCRSelected

```
// GCSPCRSelected()
// =====
// Returns TRUE if the Guarded control stack is PCR selected
// at the provided Exception level.

boolean GCSPCRSelected(bits(2) el)
    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';
    Unreachable();
    return TRUE;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPCX

```
// GCSPOPCX()
// =====
// Called to pop and compare a Guarded control stack exception return record.

GCSPOPCX()
    constant bits(64) spsr = SPSR\_ELx[];
    CheckGCSExRecord(ELR\_ELx[], spsr, X[30,64], GCSInstType\_POPCX);
    PSTATE.EXLOCK = if GetCurrentEXLOCKEN() then '1' else '0';
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPM

```
// GCSPOPM()
// =====
// Called to pop a Guarded control stack procedure return record.

bits(64) GCSPOPM()
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_LOAD, privileged);

    ptr = GetCurrentGCSPtr();
    constant bits(64) gcs_entry = Mem[ptr, 8, accdesc];

    if gcs_entry<1:0> != '00' then
        GCSDataCheckException(GCSInstType\_POPM);

    ptr = ptr + 8;
    SetCurrentGCSPtr(ptr);
    return gcs_entry;
```

## Library pseudocode for aarch64/functions/gcs/GCSPOPX

```
// GCSPOPX()
// =====
// Called to pop a Guarded control stack exception return record.

GCSPOPX()
    bits(64) ptr;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_LOAD, privileged);
    ptr = GetCurrentGCSPtr();

    // Check the lowest doubleword is correctly formatted
    constant bits(64) recorded_first_dword = Mem[ptr, 8, accdesc];
    if recorded_first_dword != Zeros(60):'1001' then
        GCSDataCheckException(GCSInstType\_POPX);

    // Ignore these loaded values, however they might have
    // faulted which is why we load them anyway
    constant bits(64) recorded_elr = Mem[ptr+8, 8, accdesc];
    constant bits(64) recorded_spsr = Mem[ptr+16, 8, accdesc];
    constant bits(64) recorded_lr = Mem[ptr+24, 8, accdesc];

    // Increment the pointer value
    ptr = ptr + 32;

    SetCurrentGCSPtr(ptr);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPUSHM

```
// GCSPUSHM()
// =====
// Called to push a Guarded control stack procedure return record.

GCSPUSHM(bits(64) value)
    AddGCSRecord(value);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSPUSHX

```
// GCSPUSHX()
// =====
// Called to push a Guarded control stack exception return record.

GCSPUSHX()
    constant bits(64) spsr = SPSR\_ELx[];
    AddGCSExRecord(ELR\_ELx[], spsr, X[30,64]);
    PSTATE.EXLOCK = '0';
    return;
```

## 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.

boolean GCSReturnValueCheckEnabled(bits(2) el)
    if UsingAArch32() then
        return FALSE;
    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';
```

## Library pseudocode for aarch64/functions/gcs/GCSSS1

```
// GCSSS1()
// =====
// Operational pseudocode for GCSSS1 instruction.

GCSSS1(bits(64) incoming_pointer)
    bits(64) outgoing_pointer, cmpoperand, operand, data;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCSSS1(privileged);
    outgoing_pointer = GetCurrentGCSPointer();
    // Valid cap entry is expected
    cmpoperand = incoming_pointer[63:12]:'000000000001';
    // In-progress cap entry should be stored if the comparison is successful
    operand = outgoing_pointer[63:3]:'101';

    data = MemAtomic(incoming_pointer, cmpoperand, operand, accdesc);
    if data == cmpoperand then
        SetCurrentGCSPointer(incoming_pointer[63:3]:'000');
    else
        GCSDDataCheckException(GCSInstType\_SS1);
    return;
```

## Library pseudocode for aarch64/functions/gcs/GCSSS2

```
// GCSSS2()
// =====
// Operational pseudocode for GCSSS2 instruction.

bits(64) GCSSS2()
    bits(64) outgoing_pointer, incoming_pointer, outgoing_value;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc_ld = CreateAccDescGCS(MemOp\_LOAD, privileged);
    constant AccessDescriptor accdesc_st = CreateAccDescGCS(MemOp\_STORE, privileged);
    incoming_pointer = GetCurrentGCSPointer();
    outgoing_value = Mem[incoming_pointer, 8, 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[outgoing_pointer, 8, accdesc_st] = outgoing_value;
        SetCurrentGCSPointer(incoming_pointer + 8);
        GCSSynchronizationBarrier();
    else
        GCSDataCheckException(GCSInstType\_SS2);
    return outgoing_pointer;
```

## Library pseudocode for aarch64/functions/gcs/GCSSSTRTrapException

```
// GCSSSTRTrapException()
// =====
// Handle a trap on GCSSSTR instruction condition.

GCSSSTRTrapException(bits(2) target_el)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;

    except = 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);
```

## Library pseudocode for aarch64/functions/gcs/GCSSSynchronizationBarrier

```
// GCSSSynchronizationBarrier()
// =====
// Barrier instruction that synchronizes Guarded Control Stack
// accesses in relation to other load and store accesses

GCSSSynchronizationBarrier();
```

## Library pseudocode for aarch64/functions/gcs/GetCurrentEXLOCKEN

```
// GetCurrentEXLOCKEN()
// =====

boolean GetCurrentEXLOCKEN()
    if Halted\(\) || Restarting\(\) then
        return FALSE;

    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';
```

## Library pseudocode for aarch64/functions/gcs/GetCurrentGCSPtr

```
// GetCurrentGCSPtr()
// =====
// Returns the value of the current Guarded control stack
// pointer register.

bits(64) GetCurrentGCSPtr()
    bits(64) ptr;

    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';
    return ptr;
```

## Library pseudocode for aarch64/functions/gcs/LoadCheckGCSRecord

```
// LoadCheckGCSRecord()
// =====
// Validates the provided address against the top entry of the
// current Guarded control stack.

bits(64) LoadCheckGCSRecord(bits(64) vaddress, GCSInstruction gcsinst_type)
    bits(64) ptr;
    bits(64) recorded_va;
    constant boolean privileged = PSTATE.EL != EL0;
    constant AccessDescriptor accdesc = CreateAccDescGCS(MemOp\_LOAD, privileged);

    ptr = GetCurrentGCSPtr();
    recorded_va = Mem[ptr, 8, accdesc];
    if GCSReturnValueCheckEnabled(PSTATE.EL) && (recorded_va != vaddress) then
        GCSDataCheckException(gcsinst_type);

    return recorded_va;
```

### Library pseudocode for aarch64/functions/gcs/SetCurrentGCSPtr

```
// SetCurrentGCSPtr()
// =====
// Writes a value to the current Guarded control stack pointer register.

SetCurrentGCSPtr(bits(64) ptr)
  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>;
  return;
```

### Library pseudocode for aarch64/functions/hacdb/HACDBS\_ERR\_REASON\_IPAF

```
constant bits(2) HACDBS_ERR_REASON_IPAF = '10';
```

### Library pseudocode for aarch64/functions/hacdb/HACDBS\_ERR\_REASON\_IPHACF

```
constant bits(2) HACDBS_ERR_REASON_IPHACF = '11';
```

### Library pseudocode for aarch64/functions/hacdb/HACDBS\_ERR\_REASON\_STRUCTF

```
constant bits(2) HACDBS_ERR_REASON_STRUCTF = '01';
```

### Library pseudocode for aarch64/functions/hacdb/IsHACDBSIRQAsserted

```
// IsHACDBSIRQAsserted()
// =====
// Returns TRUE if HACDBSIRQ is asserted, and FALSE otherwise.

boolean IsHACDBSIRQAsserted();
```



```

// ProcessHACDBSEntry()
// =====
// Process a single entry entry from the HACDBS.

ProcessHACDBSEntry()
    if !IsFeatureImplemented(FEAT_HACDBS) then return;

    if (HaveEL(EL3) && SCR_EL3.HACDBSEn == '0') || HACDBSBR_EL2.EN == '0' then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, Signal_Low);
        return;

    if HCR_EL2.VM == '0' then return;

    if (UInt(HACDBSCONS_EL2.INDEX) >= (2 ^ (UInt(HACDBSBR_EL2.SZ) + 12)) DIV 8 ||
        HACDBSCONS_EL2.ERR_REASON != '00') then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, Signal_High);
        return;
    elseif IsHACDBSIRQAsserted() then
        SetInterruptRequestLevel(InterruptID HACDBSIRQ, Signal_Low);

    constant integer hacdbbs_size = UInt(HACDBSBR_EL2.SZ);
    bits(56) baddr = HACDBSBR_EL2.BADDR : Zeros(12);
    baddr<11 + hacdbbs_size : 12> = Zeros(hacdbbs_size);

    AccessDescriptor accdesc = CreateAccDescHACDBS();

    AddressDescriptor addrdesc;
    addrdesc.paddress.address = baddr + (8 * UInt(HACDBSCONS_EL2.INDEX));
    constant bit nse2 = '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.ORGNO);
    constant bit emec = (if IsFeatureImplemented(FEAT_MEC) &&
        IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
    addrdesc.mecid = AArch64.S2TTWalkMECID(emec, accdesc.ss);

    FaultRecord fault = 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;

    PhysMemRetStatus memstatus;
    bits(64) hacdbbs_entry;
    (memstatus, hacdbbs_entry) = PhysMemRead(addrdesc, 8, accdesc);

    if IsFault(memstatus) then
        HACDBSCONS_EL2.ERR_REASON = HACDBS_ERR_REASON_STRUCTF;
        return;

    if BigEndian(accdesc.acctype) then
        hacdbbs_entry = BigEndianReverse(hacdbbs_entry);

    // 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;

    accdesc = CreateAccDescTTEUpdate(accdesc);
    AddressDescriptor ipa;
    ipa.paddress.address = hacdbbs_entry<55:12> : Zeros(12);

    constant bit nsipa = hacdbbs_entry<11>;
    constant 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);

constant boolean slaarch64 = TRUE;
constant S2TTWParams walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace,
                                                    slaarch64);

AddressDescriptor descpaddr;
TTWState walkstate;
bits(128) descriptor;
if walkparams.d128 == '1' then
    (fault, descpaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                            accdesc, 128);
else
    (fault, descpaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                    accdesc, 64);

// 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;
elsif fault.statuscode != Fault None then
    HACDBSCONS_EL2.ERR_REASON = HACDBS\_ERR\_REASON\_IPAF;
    return;

constant integer hacdbcs_level = SInt(hacdbcs_entry<3:1>);
if walkstate.level != hacdbcs_level || walkstate.contiguous == '1' then
    HACDBSCONS_EL2.ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
    return;

// 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 writeable-clean or writeable-dirty.

// Check if the descriptor is neither writeable-clean nor writeable-dirty.
if walkparams.s2pie == '1' then
    constant S2AccessControls perms = AArch64.S2ComputePermissions(walkstate.permissions,
                                                                    walkparams, accdesc);

    if perms.w == '0' && perms.w_mmu == '0' then
        HACDBSCONS_EL2.ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
        return;

// If DBM is 0, the descriptor is not writeable-clean or writeable-dirty.
elsif descriptor<51> == '0' then
    HACDBSCONS_EL2.ERR_REASON = HACDBS\_ERR\_REASON\_IPHACF;
    return;

// If the descriptor is writeable-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;

bits(128) new_descriptor = descriptor;
new_descriptor<7> = '0';

constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
if walkparams.d128 == '1' then
    (fault, -) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor, walkparams.ee,
                                           descaccess, descpaddr);
else
    (fault, -) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>, new_descriptor<63:0>,
                                           walkparams.ee, descaccess, descpaddr);

if fault.statuscode != Fault None then
    HACDBSCONS_EL2.ERR_REASON = HACDBS\_ERR\_REASON\_IPAF;
else
    HACDBSCONS_EL2.INDEX = HACDBSCONS_EL2.INDEX + 1;

return;

```



```

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(CacheOpScope opscope)
    regval = bits(64) UNKNOWN;
    AArch64.IC(regval, opscope);

// AArch64.IC()
// =====
// Perform Instruction Cache Operation.

AArch64.IC(bits(64) regval, CacheOpScope opscope)
    CacheRecord cache;

    cache.acctype = AccessType_IC;
    cache.cachetype = CacheType_Instruction;
    cache.cacheop = CacheOp_Invalidate;
    cache.opscope = opscope;

    if opscope IN {CacheOpScope_ALLU, CacheOpScope_ALLUIS} then
        ss = SecurityStateAtEL(PSTATE.EL);
        cache.cpas = CPASAtSecurityState(ss);
        if opscope == CacheOpScope_ALLUIS then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        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;
        else
            cache.is_vmid_valid = FALSE;

        if PSTATE.EL == EL0 then
            cache.is_asid_valid = TRUE;
            cache.asid = ASID[];
        else
            cache.is_asid_valid = FALSE;

        constant bits(64) vaddress = regval;
        constant boolean need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress = regval;
        cache.shareability = Shareability_NSH;
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        constant AccessDescriptor accdesc = CreateAccDescIC(cache);
        constant boolean aligned = TRUE;
        constant integer size = 0;
        AddressDescriptor memaddrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

        if IsFault(memaddrdesc) then
            memaddrdesc.fault.vaddress = regval;
            AArch64.Abort(memaddrdesc.fault);

        cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
        cache.paddress = memaddrdesc.paddress;

```

```

    CACHE\_OP(cache);
return;

```

## Library pseudocode for aarch64/functions/immediateop/ImmediateOp

```

// ImmediateOp
// =====
// Vector logical immediate instruction types.

enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
    ImmediateOp_ORR, ImmediateOp_BIC};

```

## Library pseudocode for aarch64/functions/logicalop/LogicalOp

```

// LogicalOp
// =====
// Logical instruction types.

enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};

```

## 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.

boolean AArch64.S1AMECFault(S1TTWParams walkparams, PASpace paspace, Regime regime,
    bits(N) descriptor)
{
    assert N IN {64,128};
    constant bit descriptor_amec = (if walkparams.d128 == '1' then descriptor<108>
        else descriptor<63>);

    return (walkparams.<emec,amec> == '10' &&
        regime IN {Regime\_EL2, Regime\_EL20} &&
        paspace == PAS Realm &&
        descriptor_amec == '1');
}

```

## Library pseudocode for aarch64/functions/mec/AArch64.S1DisabledOutputMECID

```

// AArch64.S1DisabledOutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is disabled.

bits(16) AArch64.S1DisabledOutputMECID(S1TTWParams walkparams, Regime regime, PASpace paspace)
{
    if walkparams.emec == '0' then
        return DEFAULT MECID;

    if ! regime IN {Regime\_EL2, Regime\_EL20, Regime\_EL10} then
        return DEFAULT MECID;

    if paspace != PAS Realm then
        return DEFAULT MECID;

    if regime == Regime\_EL10 then
        return VMECID_P_EL2.MECID;
    else
        return MECID_P0_EL2.MECID;
}

```

## Library pseudocode for aarch64/functions/mec/AArch64.S1OutputMECID

```
// AArch64.S1OutputMECID()
// =====
// Returns the output MECID when stage 1 address translation is enabled.

bits(16) AArch64.S1OutputMECID(S1TTWParams walkparams, Regime regime, VARange varange,
                                PASpace paspace, bits(N) descriptor)

    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT_MECID;

    if paspace != PAS_Realm then
        return DEFAULT_MECID;

    constant bit descriptor_amec = (if walkparams.d128 == '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;
        when Regime_EL20
            if varange == VARange_LOWER then
                if descriptor_amec == '0' then
                    return MECID_P0_EL2.MECID;
                else
                    return MECID_A0_EL2.MECID;
            else
                if descriptor_amec == '0' then
                    return MECID_P1_EL2.MECID;
                else
                    return MECID_A1_EL2.MECID;
        when Regime_EL10
            return VMECID_P_EL2.MECID;
```

## 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.

bits(16) AArch64.S1TTWalkMECID(bit emec, Regime regime, SecurityState ss)

    if emec == '0' then
        return DEFAULT_MECID;

    if ss != SS_Realm then
        return DEFAULT_MECID;

    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;
        // 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();
```

### Library pseudocode for aarch64/functions/mec/AArch64.S2OutputMECID

```
// AArch64.S2OutputMECID()
// =====
// Returns the output MECID for stage 2 address translation.

bits(16) AArch64.S2OutputMECID(S2TTWParams walkparams, PASpace paspace, bits(N) descriptor)
    assert N IN {64,128};

    if walkparams.emec == '0' then
        return DEFAULT\_MECID;

    if paspace != PAS\_Realm then
        return DEFAULT\_MECID;

    constant bit descriptor_amec = (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;
```

### 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.

bits(16) AArch64.S2TTWalkMECID(bit emec, SecurityState ss)
    if emec == '0' then
        return DEFAULT\_MECID;

    if ss != SS\_Realm then
        return DEFAULT\_MECID;

    //Stage 2 translation might later override the MECID according to AMEC configuration
    return VMECID_P_EL2.MECID;
```

### Library pseudocode for aarch64/functions/mec/DEFAULT\_MECID

```
constant bits(16) DEFAULT_MECID = Zeros(16);
```

## Library pseudocode for aarch64/functions/memory/AArch64.AccessIsTagChecked

```
// AArch64.AccessIsTagChecked()
// =====
// TRUE if a given access is tag-checked, FALSE otherwise.

boolean AArch64.AccessIsTagChecked(bits(64) vaddr, AccessDescriptor accdesc)
    assert accdesc.tagchecked;

    if UsingAArch32() then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(accdesc.el) then
        return FALSE;

    if PSTATE.TCO == '1' then
        return FALSE;

    if (Halted() && EDSCR.MA == '1' &&
        ConstrainUnpredictableBool(Unpredictable\_NODTRTAGCHK)) then
        return FALSE;

    if (IsFeatureImplemented(FEAT_MTE_STORE_ONLY) && !accdesc.write &&
        StoreOnlyTagCheckingEnabled(accdesc.el)) then
        return FALSE;

    constant boolean is_instr = FALSE;
    if (EffectiveMTX(vaddr, is_instr, PSTATE.EL) == '0' &&
        EffectiveTBI(vaddr, is_instr, PSTATE.EL) == '0') then
        return FALSE;

    if (EffectiveTCMA(vaddr, PSTATE.EL) == '1' &&
        (vaddr<59:55> == '00000' || vaddr<59:55> == '11111')) then
        return FALSE;

    return TRUE;
```

## 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.
// If the extension is disabled, treats the Allocation Tag as '0000'.

bits(64) AArch64.AddressWithAllocationTag(bits(64) address, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(PSTATE.EL) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

## 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.

boolean AArch64.AllocationTagCheck(AddressDescriptor memaddrdesc, AccessDescriptor accdesc,
                                   bits(4) ltag)
    if memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);

        return ltag == readtag;
    else
        return TRUE;
```

## Library pseudocode for aarch64/functions/memory/AArch64.AllocationTagFromAddress

```
// AArch64.AllocationTagFromAddress()
// =====
// Generate an Allocation Tag from a 64-bit value containing a Logical Address Tag.

bits(4) AArch64.AllocationTagFromAddress(bits(64) tagged_address)
    return tagged_address<59:56>;
```

## 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.

boolean AArch64.CanonicalTagCheck(AddressDescriptor memaddrdesc, bits(4) ltag)
    expected_tag = if memaddrdesc.vaddress<55> == '0' then '0000' else '1111';
    return ltag == expected_tag;
```

## Library pseudocode for aarch64/functions/memory/AArch64.CheckTag

```
// AArch64.CheckTag()
// =====
// Performs a Tag Check operation for a memory access and returns
// whether the check passed

boolean AArch64.CheckTag(AddressDescriptor memaddrdesc, AccessDescriptor accdesc, bits(4) ltag)
    if memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        return AArch64.AllocationTagCheck(memaddrdesc, accdesc, ltag);
    elsif memaddrdesc.memattrs.tags == MemTag\_CanonicallyTagged then
        return AArch64.CanonicalTagCheck(memaddrdesc, ltag);
    else
        return TRUE;
```



### Library pseudocode for aarch64/functions/memory/AArch64.IsUnprivAccessPriv

```
// AArch64.IsUnprivAccessPriv()
// =====
// Returns TRUE if an unprivileged access is privileged, and FALSE otherwise.

boolean AArch64.IsUnprivAccessPriv()
    boolean privileged;

    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;

    if IsFeatureImplemented(FEAT_UAO) && PSTATE.UAO == '1' then
        privileged = PSTATE.EL != EL0;

    return privileged;
```

### Library pseudocode for aarch64/functions/memory/AArch64.LogicalAddressTag

```
// AArch64.LogicalAddressTag()
// =====
// Extract the Logical Address Tag from an address

bits(4) AArch64.LogicalAddressTag(bits(64) vaddr)
    return vaddr<59:56>;
```

## Library pseudocode for aarch64/functions/memory/AArch64.MemSingle

```
// AArch64.MemSingle[] - non-assignment (read) form
// =====
// Perform an atomic, little-endian read of 'size' bytes.

bits(size*8) AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                                boolean aligned]
    bits(size*8) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (value, memaddrdesc, memstatus) = AArch64.MemSingleRead(address, size, accdesc, aligned);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalAbort(memstatus, accdesc.write, memaddrdesc, size, accdesc);

    return value;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccessDescriptor accdesc,
                  boolean aligned] = bits(size*8) value
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;

    (memaddrdesc, memstatus) = AArch64.MemSingleWrite(address, size, accdesc, aligned, value);

    // Check for a fault from translation or the output of translation.
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Check for external aborts.
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;
```



```

// AArch64.MemSingleRead()
// =====
// Perform an atomic, little-endian read of 'size' bytes.

(bits(size*8), AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleRead(bits(64) address,
                                                                    integer size,
                                                                    AccessDescriptor accdesc_in,
                                                                    boolean aligned)

assert size IN {1, 2, 4, 8, 16, 32};
bits(size*8) value = bits(size*8) UNKNOWN;
PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
AccessDescriptor accdesc = accdesc_in;
if IsFeatureImplemented(FEAT_LSE2) then
    constant integer quantity = MemSingleGranule();
    assert ((IsFeatureImplemented(FEAT_LS64WB) &&
        size == 32 && accdesc.acctype == AccessType_ASIMD) ||
        AllInAlignedQuantity(address, size, quantity));
else
    assert IsAligned(address, size);

// If the instruction encoding permits tag checking, confer with system register configuration
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

AddressDescriptor memaddrdesc;
memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (value, memaddrdesc, memstatus);

// Memory array access
if IsFeatureImplemented(FEAT_TME) then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
        FailTransaction(TMFailure_IMP, FALSE);

if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    constant bits(4) ltag = AArch64.LogicalAddressTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
        constant TCFTYPE tcf = AArch64.EffectiveTCF(accdesc.el, accdesc.read);
        case tcf of
            when TCFTYPE_Ignore
                // Tag Check Faults have no effect on the PE.
            when TCFTYPE_Sync
                memaddrdesc.fault.statuscode = Fault_TagCheck;
                memaddrdesc.fault.vaddress = address;
                return (value, memaddrdesc, memstatus);
            when TCFTYPE_Async
                AArch64.ReportTagCheckFault(accdesc.el, address<55>);

if SPESampleInFlight then
    constant boolean is_load = TRUE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

boolean atomic;
if IsWBShareable(memaddrdesc.memattrs) then
    atomic = TRUE;
elseif (accdesc.exclusive || accdesc.atomicop ||
    accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    atomic = TRUE;
elseif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within MemSingleGranule() byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";

if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);

```

```

    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);

elseif accdesc.acctype == AccessType ASIMD && size == 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 IsAligned(address, 32);
    accdesc.ispair = FALSE;
    for i = 0 to 3
        (memstatus, value<i*64+:64>) = PhysMemRead(memaddrdesc, 8, accdesc);
        if IsFault(memstatus) then
            return (value, memaddrdesc, memstatus);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;

elseif aligned && accdesc.ispair then
    assert size IN {8, 16};
    constant integer halfsize = size DIV 2;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        return (value, memaddrdesc, memstatus);
    value = highhalf:lowhalf;

else
    for i = 0 to size-1
        (memstatus, Elem[value, i, 8]) = PhysMemRead(memaddrdesc, 1, accdesc);
        if IsFault(memstatus) then
            return (value, memaddrdesc, memstatus);

        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;

return (value, memaddrdesc, memstatus);

```



```

// AArch64.MemSingleWrite()
// =====
// Perform an atomic, little-endian write of 'size' bytes.

(AddressDescriptor, PhysMemRetStatus) AArch64.MemSingleWrite(bits(64) address, integer size,
                                                             AccessDescriptor accdesc_in,
                                                             boolean aligned, bits(size*8) value)

assert size IN {1, 2, 4, 8, 16, 32};
AccessDescriptor accdesc = accdesc_in;
if IsFeatureImplemented(FEAT_LSE2) then
    constant integer quantity = MemSingleGranule();
    assert ((IsFeatureImplemented(FEAT_LS64WB) &&
             size == 32 && accdesc.acctype == AccessType\_ASIMD) ||
            AllInAlignedQuantity(address, size, quantity));
else
    assert IsAligned(address, size);

// If the instruction encoding permits tag checking, confer with system register configuration
// which may override this.
if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;
memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    return (memaddrdesc, memstatus);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

if IsFeatureImplemented(FEAT_TME) then
    if accdesc.transactional && !MemHasTransactionalAccess(memaddrdesc.memattrs) then
        FailTransaction(TMFailure\_IMP, FALSE);

if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
    constant bits(4) ltag = AArch64.LogicalAddressTag(address);
    if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
        constant TCFType tcf = AArch64.EffectiveTCF(accdesc.el, accdesc.read);
        case tcf of
            when TCFType\_Ignore
                // Tag Check Faults have no effect on the PE.
            when TCFType\_Sync
                memaddrdesc.fault.statuscode = Fault\_TagCheck;
                memaddrdesc.fault.vaddress = address;
                return (memaddrdesc, memstatus);
            when TCFType\_Async
                AArch64.ReportTagCheckFault(accdesc.el, address<55>);

if SPESampleInFlight then
    constant boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

boolean atomic;
if IsWBShareable(memaddrdesc.memattrs) then
    atomic = TRUE;
elsif (accdesc.exclusive || accdesc.atomicop ||
        accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
    atomic = TRUE;
elsif aligned then
    atomic = !accdesc.ispair;
else
    // Misaligned accesses within MemSingleGranule() byte aligned memory but
    // not Normal Cacheable Writeback are Atomic
    atomic = boolean IMPLEMENTATION_DEFINED "FEAT_LSE2: access is atomic";
if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);

```

```

    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);

elseif accdesc.acctype == AccessType ASIMD && size == 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 IsAligned(address, 32);
    accdesc.ispair = FALSE;
    for i = 0 to 3
        memstatus = PhysMemWrite(memaddrdesc, 8, accdesc, value<64*i+:64>);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 8;

elseif aligned && accdesc.ispair then
    assert size IN {8, 16};
    constant integer halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    <highhalf, lowhalf> = value;

    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);

    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
    if IsFault(memstatus) then
        return (memaddrdesc, memstatus);

else
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Elem[value, i, 8]);
        if IsFault(memstatus) then
            return (memaddrdesc, memstatus);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;

return (memaddrdesc, memstatus);

```



## Library pseudocode for aarch64/functions/memory/AArch64.MemTag

```
// AArch64.MemTag[] - non-assignment (read) form
// =====
// Load an Allocation Tag from memory.

bits(4) AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in]
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    constant boolean aligned = TRUE;

    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Return the granule tag if tagging is enabled.
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return tag;
    elseif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
            accdesc.tagaccess &&
            memaddrdesc.memattrs.tags == MemTag\_CanonicallyTagged) then
        return if address<55> == '0' then '0000' else '1111';
    else
        // otherwise read tag as zero.
        return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccessDescriptor accdesc_in] = bits(4) value
    assert accdesc_in.tagaccess && !accdesc_in.tagchecked;

    AddressDescriptor memaddrdesc;
    AccessDescriptor accdesc = accdesc_in;

    constant boolean aligned = IsAligned(address, TAG\_GRANULE);

    // Stores of allocation tags must be aligned
    if !aligned then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    accdesc.tagaccess = AArch64.AllocationTagAccessIsEnabled(accdesc.el);

    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, TAG\_GRANULE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Memory array access
    if accdesc.tagaccess && memaddrdesc.memattrs.tags == MemTag\_AllocationTagged then
        memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

## Library pseudocode for aarch64/functions/memory/AArch64.UnalignedAccessFaults

```
// AArch64.UnalignedAccessFaults()
// =====
// Determine whether the unaligned access generates an Alignment fault

boolean AArch64.UnalignedAccessFaults(AccessDescriptor accdesc, bits(64) address, integer size)
    if AlignmentEnforced() then
        return TRUE;
    elseif accdesc.acctype == AccessType\_GCS then
        return TRUE;
    elseif accdesc.rcw then
        return TRUE;
    elseif accdesc.ls64 then
        return TRUE;
    elseif (accdesc.exclusive || accdesc.atomicop) then
        constant integer quantity = MemSingleGranule();
        return (!IsFeatureImplemented(FEAT_LSE2) ||
                !AllInAlignedQuantity(address, size, quantity));
    elseif (accdesc.acqsc || accdesc.acqpc || accdesc.relsc) then
        if IsFeatureImplemented(FEAT_LSE2) then
            return (SCTLR_ELx[].nAA == '0' &&
                    !AllInAlignedQuantity(address, size, 16));
        else
            return TRUE;
    else
        return FALSE;
```

## 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.

boolean AddressSupportsLS64(bits(56) paddress);
```

## Library pseudocode for aarch64/functions/memory/AllInAlignedQuantity

```
// AllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean AllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    return Align((address+size)-1, alignment) == Align(address, alignment);
```

## Library pseudocode for aarch64/functions/memory/CASCompare

```
// CASCompare()
// =====
// Performs a comparison for CAS

(bits(N), boolean, bits(N)) CASCompare(bits(N) oldvalue, bits(N) comparevalue, bits(N) newvalue)
    bits(N) memresult;
    boolean cmpfail;
    bits(N) regresult;

    if oldvalue == comparevalue then
        cmpfail = FALSE;
        memresult = newvalue;
        if ConstrainUnpredictableBool(Unpredictable\_CASRETURNOLDVALUE) then
            regresult = oldvalue;
        else
            regresult = comparevalue;
    else
        cmpfail = TRUE;
        memresult = oldvalue;
        regresult = oldvalue;

    return (memresult, cmpfail, regresult);
```

## Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
    constant bits(64) sp = SP[64];
    boolean stack_align_check;
    if PSTATE.EL == EL0 then
        stack_align_check = (SCTLR_ELx[].SA0 != '0');
    else
        stack_align_check = (SCTLR_ELx[].SA != '0');

    if stack_align_check && sp != Align(sp, 16) then
        AArch64.SPAlignmentFault();

    return;
```

## Library pseudocode for aarch64/functions/memory/IsConventionalMemory

```
// IsConventionalMemory()
// =====
// Returns TRUE if the memory location is in Conventional memory, and FALSE otherwise.

boolean IsConventionalMemory(AddressDescriptor addrdesc);
```



```

// Mem[] - non-assignment (read) form
// =====
// Perform a read of 'size' bytes. The access byte order is reversed for a big-endian access.
// Instruction fetches would call AArch64.MemSingle directly.

bits(size*8) Mem[bits(64) address, integer size, AccessDescriptor accdesc_in]
    assert size IN {1, 2, 4, 8, 16, 32};
    AccessDescriptor accdesc = accdesc_in;
    bits(size * 8) value;
    // Check alignment on size of element accessed, not overall access size
    constant integer alignment = if accdesc.ispair then size DIV 2 else size;
    boolean aligned    = IsAligned(address, alignment);
    constant integer quantity = MemSingleGranule();

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);
    if accdesc.acctype == AccessType\_ASIMD && size == 16 && IsAligned(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.
        constant integer halfsize = size DIV 2;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        bits(halfsize * 8) lowhalf, highhalf;
        lowhalf = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elseif (accdesc.acctype == AccessType\_ASIMD && size == 32 &&
            accdesc.ispair && IsAligned(address, 32)) then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    elseif accdesc.acctype == AccessType\_ASIMD && size == 32 && IsAligned(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
            constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
            value<64*i+:64> = AArch64.MemSingle[blockaddress, 8, accdesc, aligned];
    elseif (IsFeatureImplemented(FEAT_LSE2) &&
            AllInAlignedQuantity(address, size, quantity)) then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    elseif accdesc.ispair && aligned then
        accdesc.ispair = FALSE;
        constant integer halfsize = size DIV 2;
        constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
        bits(halfsize * 8) lowhalf, highhalf;
        if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.highestaddressfirst then
            highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
            lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
        else
            lowhalf  = AArch64.MemSingle[address, halfsize, accdesc, aligned];
            highhalf = AArch64.MemSingle[highaddress, halfsize, accdesc, aligned];
        value = highhalf:lowhalf;
    elseif aligned then
        value = AArch64.MemSingle[address, size, accdesc, aligned];
    else
        assert size > 1;
        if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair && accdesc.highestaddressfirst then
            constant integer halfsize = size DIV 2;
            bits(halfsize * 8) lowhalf, highhalf;
            for i = 0 to halfsize-1
                constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
                // Individual byte access can be observed in any order
                Elem[highhalf, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];
            for i = 0 to halfsize-1
                constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
                // Individual byte access can be observed in any order
                Elem[lowhalf, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];

            value = highhalf:lowhalf;

```

```

else
    value<7:0> = AArch64.MemSingle[address, 1, 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.
    c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
    assert c IN {Constraint\_FAULT, Constraint\_NONE};
    if c == Constraint\_NONE then aligned = TRUE;

    for i = 1 to size-1
        constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
        Elem[value, i, 8] = AArch64.MemSingle[byteaddress, 1, accdesc, aligned];

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(value);

return value;

// Mem[] - assignment (write) form
// =====
// Perform a write of 'size' bytes. The byte order is reversed for a big-endian access.
Mem[bits(64) address, integer size, AccessDescriptor accdesc_in] = bits(size*8) value_in
bits(size*8) value = value_in;
AccessDescriptor accdesc = accdesc_in;

// Check alignment on size of element accessed, not overall access size
constant integer alignment = if accdesc.ispair then size DIV 2 else size;
boolean aligned = IsAligned(address, alignment);
constant integer quantity = MemSingleGranule();

if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
    constant FaultRecord fault = AlignmentFault(accdesc, address);
    AArch64.Abort(fault);

if BigEndian(accdesc.acctype) then
    value = BigEndianReverse(value);
if accdesc.acctype == AccessType\_ASIMD && size == 16 && IsAligned(address, 8) then
    constant integer halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
    // 64-bit aligned.
    <highhalf, lowhalf> = value;
    constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
    AArch64.MemSingle[address, halfsize, accdesc, aligned] = lowhalf;
    AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
elseif (accdesc.acctype == AccessType\_ASIMD && size == 32 &&
        accdesc.ispair && IsAligned(address, 32)) then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
elseif accdesc.acctype == AccessType\_ASIMD && size == 32 && IsAligned(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
        constant bits(64) blockaddress = AddressIncrement(address, i*8, accdesc);
        AArch64.MemSingle[blockaddress, 8, accdesc, aligned] = value<64*i+:64>;
elseif (IsFeatureImplemented(FEAT_LSE2) &&
        AllInAlignedQuantity(address, size, quantity)) then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
elseif accdesc.ispair && aligned then
    constant integer halfsize = size DIV 2;
    bits(halfsize*8) lowhalf, highhalf;
    accdesc.ispair = FALSE;
    <highhalf, lowhalf> = value;
    constant bits(64) highaddress = AddressIncrement(address, halfsize, accdesc);
    if IsFeatureImplemented(FEAT_LRCP3) && accdesc.highestaddressfirst then
        AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;

```

```

    AArch64.MemSingle[address,      halfsize, accdesc, aligned] = lowhalf;
else
    AArch64.MemSingle[address,      halfsize, accdesc, aligned] = lowhalf;
    AArch64.MemSingle[highaddress, halfsize, accdesc, aligned] = highhalf;
elsif aligned then
    AArch64.MemSingle[address, size, accdesc, aligned] = value;
else
    assert size > 1;
    if IsFeatureImplemented(FEAT_LRCPC3) && accdesc.ispair && accdesc.highestaddressfirst then
        constant integer halfsize = size DIV 2;
        bits(halfsize*8) lowhalf, highhalf;
        <highhalf, lowhalf> = value;
        for i = 0 to halfsize-1
            constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
            // Individual byte access can be observed in any order
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[highhalf, i, 8];
        for i = 0 to halfsize-1
            constant bits(64) byteaddress = AddressIncrement(address, halfsize + i, accdesc);
            // Individual byte access can be observed in any order, but implies observability
            // of highhalf
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[lowhalf, i, 8];
    else
        AArch64.MemSingle[address, 1, 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.

        c = ConstrainUnpredictable(Unpredictable DEVPAGE2);
        assert c IN {Constraint\_FAULT, Constraint\_NONE};
        if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            constant bits(64) byteaddress = AddressIncrement(address, i, accdesc);
            AArch64.MemSingle[byteaddress, 1, accdesc, aligned] = Elem[value, i, 8];
return;

```





```

// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, bits(size) cmpoperand, bits(size) operand,
    AccessDescriptor accdesc_in)
    assert accdesc_in.atomicop;

    constant integer bytes = size DIV 8;
    assert bytes IN {1, 2, 4, 8, 16};

    bits(size) newvalue;
    bits(size) oldvalue;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // MMU or MPU lookup
    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
        aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable\_Atomic\_NOP)) then
        return bits(size) UNKNOWN;

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked &&
        (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
        !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            accdesc.write = FALSE; // Tag Check Fault on a read
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, 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);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    bits(size) retvalue = oldvalue;
    if accdesc.acctype == AccessType\_FP then
        newvalue = MemAtomicFP(accdesc.modop, oldvalue, operand);
    else
        (newvalue, cmpfail, retvalue) = MemAtomicInt(accdesc.modop, oldvalue, operand, cmpoperand);

    boolean requirewrite = !cmpfail || ConstrainUnpredictableBool(Unpredictable\_WRITEFAILEDCAS);

```

```

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\_STOREONLYTAGCHECKEDCAS);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            AArch64.TagCheckFault(address, accdesc);

if requirewrite then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse(newvalue);
    memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

if SPESampleInFlight then
    constant boolean is_load = FALSE;
    SPESampleLoadStore(is_load, accdesc, memaddrdesc);

// 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;

```

## Library pseudocode for aarch64/functions/memory/MemAtomicFP

```

// MemAtomicFP()
// =====
// Performs FP Atomic operation

bits(N) MemAtomicFP(MemAtomicOp modop, bits(N) op1, bits(N) op2)
    FPCR\_Type fpcr = FPCR;
    constant boolean altfp = FALSE;
    constant boolean fpexc = FALSE;
    fpcr.<AH,DN> = '01';
    fpcr.FZ = fpcr.FZ OR fpcr.FIZ; // Treat FPCR.FIZ as equivalent to FPCR.FZ
    bits(N) result;

    case modop of
        when MemAtomicOp\_FPADD result = FPAdd(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_FPMAX result = FPMAX(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_FPMIN result = FPMIN(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_FPMAXNM result = FPMAXNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_FPMINNM result = FPMINNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFADD result = BFAdd(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFMAX result = BFMAX(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_BFMIN result = BFMIN(op1, op2, fpcr, altfp, fpexc);
        when MemAtomicOp\_BFMAXNM result = BFMAXNum(op1, op2, fpcr, fpexc);
        when MemAtomicOp\_BFMINNM result = BFMINNum(op1, op2, fpcr, fpexc);
    return result;

```

## Library pseudocode for aarch64/functions/memory/MemAtomicInt

```
// MemAtomicInt()
// =====
// Performs Integer Atomic operation

(bits(N), boolean, bits(N)) MemAtomicInt(MemAtomicOp modop, bits(N) op1, bits(N) op2, bits(N) cmpop)
    bits(N) result;
    boolean cmpfail = FALSE;
    bits(N) retvalue = op1;

    case modop of
        when MemAtomicOp\_ADD          result = op1 + op2;
        when MemAtomicOp\_BIC          result = op1 AND NOT(op2);
        when MemAtomicOp\_EOR          result = op1 EOR 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(op1, cmpop, op2);
        when MemAtomicOp\_GCSSS1      (result, cmpfail, retvalue) = CASCompare(op1, cmpop, op2);
    return (result, cmpfail, retvalue);
```



```

// MemAtomicRCW()
// =====
// Perform a single-copy-atomic access with Read-Check-Write operation

(bits(4), bits(size)) MemAtomicRCW(bits(64) address, bits(size) cmpoperand, bits(size) operand,
                                   AccessDescriptor accdesc_in)

    assert accdesc_in.atomicop;
    assert accdesc_in.rcw;

    constant integer bytes = size DIV 8;
    assert bytes IN {8, 16};

    bits(4) nzcw;
    bits(size) oldvalue;
    bits(size) newvalue;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, bytes);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, bytes) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // MMU or MPU lookup
    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                    aligned, bytes);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    if (!IsWBShareable(memaddrdesc.memattrs) &&
        ConstrainUnpredictableBool(Unpredictable\_Atomic\_NOP)) then
        return (bits(4) UNKNOWN, bits(size) UNKNOWN);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), bytes);

    // For Store-only Tag checking, the tag check is performed on the store.
    if (IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked &&
        (!IsFeatureImplemented(FEAT_MTE_STORE_ONLY) ||
         !StoreOnlyTagCheckingEnabled(accdesc.el))) then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            accdesc.write = FALSE; // Tag Check Fault on a read
            AArch64.TagCheckFault(address, accdesc);

    // All observers in the shareability domain observe the following load and store atomically.
    PhysMemRetStatus memstatus;
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, bytes, 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);
    if BigEndian(accdesc.acctype) then
        oldvalue = BigEndianReverse(oldvalue);

    boolean cmpfail = FALSE;
    bits(size) retvalue = 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(oldvalue, cmpoperand, operand);

if cmpfail then
    nzcw = '1010'; // N = 1 indicates compare failure
else
    nzcw = RCWCheck(retvalue, newvalue, accdesc.rcws);

// If RCWCheck() passes, it returns nzcw == '0010'
boolean requirewrite = ((nzcw == '0010') ||
    (accdesc.modop == MemAtomicOp\_CAS &&
        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\_STOREONLYTAGCHECKED\_CAS);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            AArch64.TagCheckFault(address, accdesc);

if requirewrite then
    if BigEndian(accdesc.acctype) then
        newvalue = BigEndianReverse(newvalue);

    memstatus = PhysMemWrite(memaddrdesc, bytes, accdesc, newvalue);

    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, bytes, accdesc);

// 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);

```



```

// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccessDescriptor accdesc_in)
    bits(512) data;
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            AArch64.TagCheckFault(address, accdesc);

    boolean byte_atomic = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType Device ||
        (memaddrdesc.memattrs.inner.attrs == MemAttr NC &&
         memaddrdesc.memattrs.outer.attrs == MemAttr NC)) &&
        !AddressSupportsLS64(memaddrdesc.paddress.address)) then
        c = 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.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;
    elseif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if boolean IMPLEMENTATION_DEFINED "LD64B faults to iWBowB non-Conventional memory" then
            // Generate a Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;

    PhysMemRetStatus memstatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to size-1
            (memstatus, Elem[data, i, 8]) = PhysMemRead(memaddrdesc, 1, accdesc);
            if IsFault(memstatus) then
                HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    else
        (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);

    return data;

```



## 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.

integer MemSingleGranule()
    size = integer IMPLEMENTATION_DEFINED "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;
```



```

// MemStore64B()
// =====
// Performs an atomic 64-byte store to a given virtual address. Function does
// not return the status of the store.

MemStore64B(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            AArch64.TagCheckFault(address, accdesc);

    boolean byte_atomic = FALSE;
    if ((memaddrdesc.memattrs.memtype == MemType Device ||
        (memaddrdesc.memattrs.inner.attrs == MemAttr NC &&
         memaddrdesc.memattrs.outer.attrs == MemAttr NC)) &&
        !AddressSupportsLS64(memaddrdesc.paddress.address)) then
        c = 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.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;
    elseif IsWBShareable(memaddrdesc.memattrs) && !IsConventionalMemory(memaddrdesc) then
        if boolean IMPLEMENTATION_DEFINED "ST64B faults to iWBWB non-Conventional memory" then
            // Generate a Data Abort reported using the DFSC code of 110101.
            constant FaultRecord fault = ExclusiveFault(accdesc, address);
            AArch64.Abort(fault);
        else
            byte_atomic = TRUE;

    PhysMemRetStatus memstatus;
    if byte_atomic then
        // Accesses are not single-copy atomic above the byte level.
        for i = 0 to size-1
            memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Elem[value, i, 8]);
            if IsFault(memstatus) then
                HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
    else
        memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return;

```

## 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.

bits(64) MemStore64BWithRet(bits(64) address, bits(512) value, AccessDescriptor accdesc_in)
    constant integer size = 64;
    AccessDescriptor accdesc = accdesc_in;
    constant boolean aligned = IsAligned(address, size);

    if !aligned && AArch64.UnalignedAccessFaults(accdesc, address, size) then
        constant FaultRecord fault = AlignmentFault(accdesc, address);
        AArch64.Abort(fault);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    constant AddressDescriptor memaddrdesc = AArch64.TranslateAddress(address, accdesc,
                                                                    aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(memaddrdesc.fault);
        return ZeroExtend('1', 64);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ltag) then
            AArch64.TagCheckFault(address, accdesc);
            return ZeroExtend('1', 64);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);

    return memstatus.store64bstatus;
```

## 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.

bits(64) MemStore64BWithRetStatus();
```

## Library pseudocode for aarch64/functions/memory/NVMem

```
// NVMem[] - non-assignment form
// =====
// 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(VNCR_EL2.BADDR : Offset<11:0>, 64) 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.

bits(64) NVMem[integer offset]
    assert offset > 0;
    constant integer size = 64;
    return NVMem[offset, size];

bits(N) NVMem[integer offset, integer N]
    assert offset > 0;
    assert N IN {64,128};
    constant bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_LOAD);
    return Mem[address, N DIV 8, accdesc];

// NVMem[] - assignment form
// =====
// 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(VNCR_EL2.BADDR : Offset<11:0>, 64) 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.

NVMem[integer offset] = bits(64) value
    assert offset > 0;
    constant integer size = 64;
    NVMem[offset, size] = value;
    return;

NVMem[integer offset, integer N] = bits(N) value
    assert offset > 0;
    assert N IN {64,128};
    constant bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    constant AccessDescriptor accdesc = CreateAccDescNV2(MemOp_STORE);
    Mem[address, N DIV 8, accdesc] = value;
    return;
```

## 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.

(PhysMemRetStatus, bits(4)) PhysMemTagRead(AddressDescriptor desc, AccessDescriptor accdesc);
```

## 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.

PhysMemRetStatus PhysMemTagWrite(AddressDescriptor desc, AccessDescriptor accdesc, bits (4) value);
```

## Library pseudocode for aarch64/functions/memory/StoreOnlyTagCheckingEnabled

```
// StoreOnlyTagCheckingEnabled()
// =====
// Returns TRUE if loads executed at the given Exception level are Tag unchecked.

boolean StoreOnlyTagCheckingEnabled(bits (2) el)
    assert IsFeatureImplemented(FEAT_MTE_STORE_ONLY);
    bit tcso;

    case el of
        when EL0
            if !ELIsInHost(el) then
                tcso = SCTLR_EL1.TCSO0;
            else
                tcso = SCTLR_EL2.TCSO0;
        when EL1
            tcso = SCTLR_EL1.TCSO;
        when EL2
            tcso = SCTLR_EL2.TCSO;
        otherwise
            tcso = SCTLR_EL3.TCSO;

    return tcso == '1';
```

## Library pseudocode for aarch64/functions/mops/ArchMaxMOPSBBlockSize

```
// ArchMaxMOPSBBlockSize
// =====
// Maximum number of bytes CPY/SET instructions can use

constant integer ArchMaxMOPSBBlockSize = 0x7FFF_FFFF_FFFF_FFFF;
```

## Library pseudocode for aarch64/functions/mops/ArchMaxMOPSCPysize

```
// ArchMaxMOPSCPysize
// =====
// Maximum number of bytes CPY instructions can use

constant integer ArchMaxMOPSCPysize = 0x007F_FFFF_FFFF_FFFF;
```

## Library pseudocode for aarch64/functions/mops/ArchMaxMOPSETGSize

```
// ArchMaxMOPSETGSize
// =====
// Maximum number of bytes SETG instructions can use

constant integer ArchMaxMOPSETGSize = 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.

boolean CPYFOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPYF* instructions use Option A";
```

### Library pseudocode for aarch64/functions/mops/CPYOptionA

```
// CPYOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY* instructions, and FALSE otherwise.

boolean CPYOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY* instructions use Option A";
```

### Library pseudocode for aarch64/functions/mops/CPYParams

```
// CPYParams
// =====

type CPYParams is (
    MOPSSstage stage,
    boolean implements_option_a,
    boolean forward,
    integer cpysize,
    integer stagecpysize,
    bits(64) toaddress,
    bits(64) fromaddress,
    bits(4) nzcw,
    integer n,
    integer d,
    integer s
)
```

### 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.

integer CPYPostSizeChoice(CPYParams memcpy);
```

### 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.

integer CPYPreSizeChoice(CPYParams memcpy);
```

### 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.

MOPSSBlockSize CPYSizeChoice(CPYParams memcpy);
```

## Library pseudocode for aarch64/functions/mops/CheckCPYConstrainedUnpredictable

```
// CheckCPYConstrainedUnpredictable()
// =====
// Check for CONSTRAINED UNPREDICTABLE behaviour in the CPY* and CPYF* instructions.

CheckCPYConstrainedUnpredictable(integer n, integer d, integer s)
    if (s == n || s == d || n == d) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable\_MOPSOVERLAP);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP};
        case c of
            when Constraint\_UNDEF            UNDEFINED;
            when Constraint\_NOP                ExecuteAsNOP();

    if (d == 31 || s == 31 || n == 31) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable\_MOPS\_R31);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP};
        case c of
            when Constraint\_UNDEF            UNDEFINED;
            when Constraint\_NOP                ExecuteAsNOP();
```

## Library pseudocode for aarch64/functions/mops/CheckMOPSEnabled

```
// CheckMOPSEnabled()
// =====
// Check for EL0 and EL1 access to the CPY* and SET* instructions.

CheckMOPSEnabled()
    if (PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && !IsInHost() && SCTLR_EL1.MSCEn == '0' then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

## 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.

CheckMemCpyParams(CPYParams memcpy, bits(4) options)
    constant boolean from_epilogue = memcpy.stage == MOPSStage\_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memcpy.stagecpysize != 0 || MemStageCpyZeroSizeCheck()) &&
        (memcpy.cpysize != 0 || MemCpyZeroSizeCheck())) then
        constant boolean using_option_a = memcpy.nzcv<1> == '0';
        if memcpy.implements_option_a != using_option_a then
            constant boolean wrong_option = TRUE;
            MismatchedMemCpyException(memcpy, options, wrong_option);

    // Check if the parameters to this instruction are valid.
    if memcpy.stage == MOPSStage\_Main then
        if MemCpyParametersIllformedM(memcpy) then
            constant boolean wrong_option = FALSE;
            MismatchedMemCpyException(memcpy, options, wrong_option);
    else
        constant integer postsize = CPYPostSizeChoice(memcpy);
        if memcpy.cpysize != postsize || MemCpyParametersIllformedE(memcpy) then
            constant boolean wrong_option = FALSE;
            MismatchedMemCpyException(memcpy, options, wrong_option);

    return;
```



## 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.

CheckMemSetParams(SETParams memset, bits(2) options)
    constant boolean from_epilogue = memset.stage == MOPSStage\_Epilogue;

    // Check if this version is consistent with the state of the call.
    if ((memset.stagesetsize != 0 || MemStageSetZeroSizeCheck()) &&
        (memset.setsize != 0 || MemSetZeroSizeCheck())) then
        constant boolean using_option_a = memset.nzcv<1> == '0';
        if memset.implements_option_a != using_option_a then
            constant boolean wrong_option = TRUE;
            MismatchedMemSetException(memset, options, wrong_option);

    // Check if the parameters to this instruction are valid.
    if memset.stage == MOPSStage\_Main then
        if MemSetParametersIllformedM(memset) then
            constant boolean wrong_option = FALSE;
            MismatchedMemSetException(memset, options, wrong_option);
        else
            constant integer postsize = SETPostSizeChoice(memset);
            if memset.setsize != postsize || MemSetParametersIllformedE(memset) then
                constant boolean wrong_option = FALSE;
                MismatchedMemSetException(memset, options, wrong_option);

    return;
```

## Library pseudocode for aarch64/functions/mops/CheckSETConstrainedUnpredictable

```
// CheckSETConstrainedUnpredictable()
// =====
// Check for CONSTRAINED UNPREDICTABLE behaviour in the SET* and SETG* instructions.

CheckSETConstrainedUnpredictable(integer n, integer d, integer s)
    if (s == n || s == d || n == d) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable\_MOPSOVERLAP);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP};
        case c of
            when Constraint\_UNDEF      UNDEFINED;
            when Constraint\_NOP      ExecuteAsNOP();

    if (d == 31 || n == 31) then
        constant Constraint c = ConstrainUnpredictable(Unpredictable\_MOPS\_R31);
        assert c IN {Constraint\_UNDEF, Constraint\_NOP};
        case c of
            when Constraint\_UNDEF      UNDEFINED;
            when Constraint\_NOP      ExecuteAsNOP();
```

## 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.

boolean IsMemCpyForward(CPYParams memcpy)
    boolean forward;

    // Check for overlapping cases
    if ((UInt(memcpy.fromaddress<55:0>) > UInt(memcpy.toaddress<55:0>)) &&
        (UInt(memcpy.fromaddress<55:0>) < UInt(ZeroExtend(memcpy.toaddress<55:0>, 64) +
            memcpy.cpysize))) then
        forward = TRUE;

    elsif ((UInt(memcpy.fromaddress<55:0>) < UInt(memcpy.toaddress<55:0>)) &&
        (UInt(ZeroExtend(memcpy.fromaddress<55:0>, 64) + memcpy.cpysize) >
            UInt(memcpy.toaddress<55:0>))) then
        forward = FALSE;

    // Non-overlapping case
    else
        forward = boolean IMPLEMENTATION_DEFINED "CPY in the forward direction";

    return forward;
```

## Library pseudocode for aarch64/functions/mops/MOPSBlockSize

```
// MOPSBlockSize
// =====

type MOPSBlockSize = integer;
```

## Library pseudocode for aarch64/functions/mops/MOPSStage

```
// MOPSStage
// =====

enumeration MOPSStage { 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.

integer MaxBlockSizeCopiedBytes()
    return integer IMPLEMENTATION_DEFINED "Maximum bytes used in a single block of a copy";
```



```

// 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.

(integer, boolean, AddressDescriptor, PhysMemRetStatus) MemCpyBytes(bits(64) toaddress,
                                                                    bits(64) fromaddress,
                                                                    boolean forward,
                                                                    MOPSBBlockSize bytes,
                                                                    AddressDescriptor raccdesc,
                                                                    AddressDescriptor waccdesc)

AddressDescriptor rmemaddrdesc; // AddressDescriptor for reads
PhysMemRetStatus rmemstatus; // PhysMemRetStatus for writes
rmemaddrdesc.fault = NoFault();
rmemstatus.statuscode = Fault_None;

AddressDescriptor wmemaddrdesc; // AddressDescriptor for writes
PhysMemRetStatus wmemstatus; // PhysMemRetStatus for writes
wmemaddrdesc.fault = NoFault();
wmemstatus.statuscode = Fault_None;

bits(8*bytes) value;
constant boolean aligned = TRUE;

if forward then
    integer read = 0; // Bytes read
    integer write = 0; // Bytes written

    // Read until all bytes are read or until a fault is encountered.
    while read < bytes && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus) do
        (value<8 * read +:8>, rmemaddrdesc, rmemstatus) = AArch64.MemSingleRead(
                                                                    fromaddress + read, 1,
                                                                    raccdesc, aligned);

        read = read + 1;

    // Ensure no UNKNOWN data is written.
    if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
        read = read - 1;

    // Write all bytes that were read or until a fault is encountered.
    while write < read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus) do
        (wmemaddrdesc, wmemstatus) = AArch64.MemSingleWrite(toaddress + write, 1,
                                                            waccdesc, aligned,
                                                            value<8 * write +:8>);

        write = write + 1;

    // Check all bytes were written.
    if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
        constant boolean fault_on_write = TRUE;
        return (write - 1, fault_on_write, wmemaddrdesc, wmemstatus);

    // Check all bytes were read.
    if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
        constant boolean fault_on_write = FALSE;
        return (read, fault_on_write, rmemaddrdesc, rmemstatus);

else
    integer read = bytes; // Bytes to read
    integer write = bytes; // Bytes to write

    // Read until all bytes are read or until a fault is encountered.
    while read > 0 && !IsFault(rmemaddrdesc) && !IsFault(rmemstatus) do
        read = read - 1;
        (value<8 * read +:8>, rmemaddrdesc, rmemstatus) = AArch64.MemSingleRead(
                                                                    fromaddress + read, 1,
                                                                    raccdesc, aligned);

    // Ensure no UNKNOWN data is written.

```

```

if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
    read = read + 1;

// Write all bytes that were read or until a fault is encountered.
while write > read && !IsFault(wmemaddrdesc) && !IsFault(wmemstatus) do
    write = write - 1;
    (wmemaddrdesc, wmemstatus) = AArch64.MemSingleWrite(toaddress + write, 1,
                                                    waccdesc, aligned,
                                                    value<8 * write +:8>);

// Check all bytes were written.
if IsFault(wmemaddrdesc) || IsFault(wmemstatus) then
    constant boolean fault_on_write = TRUE;
    return (bytes - (write + 1), fault_on_write, wmemaddrdesc, wmemstatus);

// Check all bytes were read.
if IsFault(rmemaddrdesc) || IsFault(rmemstatus) then
    constant boolean fault_on_write = FALSE;
    return (bytes - read, fault_on_write, rmemaddrdesc, rmemstatus);

// Return any AddressDescriptor and PhysMemRetStatus.
return (bytes, FALSE, wmemaddrdesc, wmemstatus);

```

### 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.

boolean MemCpyParametersIllformedE(CPYParams memcpy);

```

### 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 CPYM* instruction for this implementation given the parameters of the destination,
// source and size of the copy.

boolean MemCpyParametersIllformedM(CPYParams memcpy);

```

## Library pseudocode for aarch64/functions/mops/MemCpyStageSize

```
// MemCpyStageSize()
// =====
// Returns the number of bytes copied by the given stage of a CPY* or CPYF* instruction.

integer MemCpyStageSize(CPYParams memcpy)
    integer stagecpysize;

    if memcpy.stage == MOPSStage\_Prologue then
        // IMP DEF selection of the amount covered by pre-processing.
        stagecpysize = CPYPreSizeChoice(memcpy);
        assert stagecpysize == 0 || (stagecpysize < 0) == (memcpy.cpysize < 0);

        if memcpy.cpysize > 0 then
            assert stagecpysize <= memcpy.cpysize;
        else
            assert stagecpysize >= memcpy.cpysize;

    else
        constant integer postsize = CPYPostSizeChoice(memcpy);
        assert postsize == 0 || (postsize < 0) == (memcpy.cpysize < 0);

        if memcpy.stage == MOPSStage\_Main then
            stagecpysize = memcpy.cpysize - postsize;
        else
            stagecpysize = postsize;

    return stagecpysize;
```

## Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// MemCpyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.

boolean MemCpyZeroSizeCheck()
    return boolean IMPLEMENTATION_DEFINED "Implementation option is checked with a cpysize of 0";
```

## 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.

(integer, AddressDescriptor, PhysMemRetStatus) MemSetBytes(bits(64) toaddress, bits(8) data,
MOPSBlockSize bytes,
AccessDescriptor accdesc)

AddressDescriptor memaddrdesc;
PhysMemRetStatus memstatus;
memaddrdesc.fault    = NoFault();
memstatus.statuscode = Fault\_None;

constant boolean aligned = TRUE;
integer write    = 0;                                // Bytes written

// Write until all bytes are written or a fault is encountered.
while write < bytes && !IsFault(memaddrdesc) && !IsFault(memstatus) do
    (memaddrdesc, memstatus) = AArch64.MemSingleWrite(toaddress + write, 1, accdesc,
                                                    aligned, data);

    write = write + 1;

// Check all bytes were written.
if IsFault(memaddrdesc) || IsFault(memstatus) then
    return (write - 1, memaddrdesc, memstatus);

return (bytes, memaddrdesc, memstatus);
```

## 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.

boolean MemSetParametersIllformedE(SETParams memset);
```

## 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.

boolean MemSetParametersIllformedM(SETParams memset);
```

## Library pseudocode for aarch64/functions/mops/MemSetStageSize

```
// MemSetStageSize()
// =====
// Returns the number of bytes set by the given stage of a SET* or SETG* instruction.

integer MemSetStageSize(SETParams memset)
    integer stagesetsize;

    if memset.stage == MOPSStage\_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';

        if memset.setsize > 0 then
            assert stagesetsize <= memset.setsize;
        else
            assert stagesetsize >= memset.setsize;

    else
        constant integer postsize = SETPostSizeChoice(memset);
        assert postsize == 0 || (postsize < 0) == (memset.setsize < 0);
        if memset.is_setg then assert postsize<3:0> == '0000';

        if memset.stage == MOPSStage\_Main then
            stagesetsize = memset.setsize - postsize;
        else
            stagesetsize = postsize;

    return stagesetsize;
```

## Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// MemSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a set of size zero remaining.

boolean MemSetZeroSizeCheck()
    return boolean IMPLEMENTATION_DEFINED "Implementation option is checked with a setsize of 0";
```

## Library pseudocode for aarch64/functions/mops/MemStageCpyZeroSizeCheck

```
// MemStageCpyZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a stage copy of size zero remaining.

boolean MemStageCpyZeroSizeCheck()
    return (boolean IMPLEMENTATION_DEFINED
            "Implementation option is checked with a stage cpysize of 0");
```

## Library pseudocode for aarch64/functions/mops/MemStageSetZeroSizeCheck

```
// MemStageSetZeroSizeCheck()
// =====
// Returns TRUE if the implementation option is checked on a stage set of size zero remaining.

boolean MemStageSetZeroSizeCheck()
    return (boolean IMPLEMENTATION_DEFINED
            "Implementation option is checked with a stage setsize of 0");
```



## Library pseudocode for aarch64/functions/mops/MismatchedCpySetTargetEL

```
// MismatchedCpySetTargetEL()
// =====
// Return the target exception level for an Exception_MemCpyMemSet.

bits(2) MismatchedCpySetTargetEL()
    bits(2) target_el;

    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;

    return target_el;
```

## 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.

MismatchedMemCpyException(CPYParams memcpy, bits(4) options, boolean wrong_option)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    constant bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = 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> = if wrong_option then '1' else '0';
    except.syndrome.iss<16> = if memcpy.implements_option_a then '1' else '0';
    // 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);
```

## 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.

MismatchedMemSetException(SETParams memset, bits(2) options, boolean wrong_option)
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    constant integer vect_offset = 0x0;
    constant bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord except = 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 == MOPSTage\_Epilogue then '1' else '0';
    except.syndrome.iss<17> = if wrong_option then '1' else '0';
    except.syndrome.iss<16> = if memset.implements_option_a then '1' else '0';
    // 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);
```

## Library pseudocode for aarch64/functions/mops/SETGOptionA

```
// SETGOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SETG* instructions, and FALSE otherwise.

boolean SETGOptionA()
    return boolean IMPLEMENTATION_DEFINED "SETG* instructions use Option A";
```

## Library pseudocode for aarch64/functions/mops/SETOptionA

```
// SETOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// SET* instructions, and FALSE otherwise.

boolean SETOptionA()
    return boolean IMPLEMENTATION_DEFINED "SET* instructions use Option A";
```

## Library pseudocode for aarch64/functions/mops/SETParams

```
// SETParams
// =====

type SETParams is (
    MOPSTage stage,
    boolean implements_option_a,
    boolean is_setg,
    integer setsize,
    integer stagesetsize,
    bits(64) toaddress,
    bits(4) nzcvc,
    integer n,
    integer d,
    integer s
)
```

### 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.

integer SETPostSizeChoice(SETParams memset);
```

### 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.

integer SETPreSizeChoice(SETParams memset);
```

### 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.

MOPSBlockSize SETSizeChoice(SETParams memset, integer alignsize);
```

## 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.

UpdateCpyRegisters(CPYParams memcpy, boolean fault, integer copied)
    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
                    // cpysize is negative.
                    constant integer cpysize = memcpy.cpysize + copied;
                    X[memcpy.n, 64] = (0 - cpysize)<63:0>;
                    X[memcpy.d, 64] = memcpy.toaddress + cpysize;
                    X[memcpy.s, 64] = memcpy.fromaddress + cpysize;

                else
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

            else
                if memcpy.forward then
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;
                    X[memcpy.d, 64] = memcpy.toaddress + copied;
                    X[memcpy.s, 64] = memcpy.fromaddress + copied;

                else
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

        else
            if memcpy.implements_option_a then
                if memcpy.forward then
                    X[memcpy.n, 64] = (memcpy.cpysize + copied)<63:0>;
                else
                    X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

            else
                X[memcpy.n, 64] = (memcpy.cpysize - copied)<63:0>;

                if memcpy.forward then
                    X[memcpy.d, 64] = memcpy.toaddress + copied;
                    X[memcpy.s, 64] = memcpy.fromaddress + copied;
                else
                    X[memcpy.d, 64] = memcpy.toaddress - copied;
                    X[memcpy.s, 64] = memcpy.fromaddress - copied;

    else
        X[memcpy.n, 64] = memcpy.cpysize<63:0>;
        if memcpy.stage == MOPSSStage Prologue || !memcpy.implements_option_a then
            X[memcpy.d, 64] = memcpy.toaddress;
            X[memcpy.s, 64] = memcpy.fromaddress;

    return;
```

## 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.

UpdateSetRegisters(SETParams memset, boolean fault, integer memory_set)
    if fault then
        // Undo any formatting of the input parameters performed in the prologue.
        if memset.stage == MOPSSStage\_Prologue then
            if memset.implements_option_a then
                // setsize is negative.
                constant integer setsize = memset.setsize + memory_set;
                X[memset.n, 64] = (0 - setsize)<63:0>;
                X[memset.d, 64] = memset.toaddress + setsize;
            else
                X[memset.n, 64] = (memset.setsize - memory_set)<63:0>;
                X[memset.d, 64] = memset.toaddress + memory_set;

        else
            if memset.implements_option_a then
                X[memset.n, 64] = (memset.setsize + memory_set)<63:0>;
            else
                X[memset.n, 64] = (memset.setsize - memory_set)<63:0>;
                X[memset.d, 64] = memset.toaddress + memory_set;

    else
        X[memset.n, 64] = memset.setsize<63:0>;
        if memset.stage == MOPSSStage\_Prologue || !memset.implements_option_a then
            X[memset.d, 64] = memset.toaddress;

    return;
```

## Library pseudocode for aarch64/functions/movewideop/MoveWideOp

```
// MoveWideOp
// =====
// Move wide 16-bit immediate instruction types.

enumeration MoveWideOp {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.

boolean MoveWidePreferred(bit sf, bit immN, bits(6) imms, bits(6) immr)
    constant integer s = UInt(imms);
    constant integer r = UInt(immr);
    constant integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && (immN:imms) != '1xxxxxx' then
        return FALSE;
    if sf == '0' && (immN:imms) != '00xxxxx' then
        return FALSE;

    // 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);

    // 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));

    return FALSE;
```

## 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.

bits(64) AddPAC(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data)
    constant boolean use_modifier2 = FALSE;
    return InsertPAC(ptr, modifier, Zeros(64), use_modifier2, K, data);
```

## 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.

bits(64) AddPAC2(bits(64) ptr, bits(64) modifier1, bits(64) modifier2, bits(128) K, boolean data)
    constant boolean use_modifier2 = TRUE;
    return InsertPAC(ptr, modifier1, modifier2, use_modifier2, K, data);
```



```

// 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.

bits(64) InsertPAC(bits(64) ptr, bits(64) modifier, bits(64) modifier2, boolean use_modifier2,
    bits(128) K, boolean data)

    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    bit bit55;
    constant boolean isgeneric = FALSE;
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant boolean mtX = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    constant integer top_bit = if tbi then 55 else 63;
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
        TranslationRegime(PSTATE.EL) == Regime EL3 &&
        AArch64.IASize(TCR_EL3.T0SZ) > 52);
    constant boolean is_VA_56bit = (TranslationRegime(PSTATE.EL) == Regime EL3 &&
        AArch64.IASize(TCR_EL3.T0SZ) == 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 1xxxxxxx0... 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>;
            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>;
        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>;
            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>;
        else selbit = if tbi then ptr<55> else ptr<63>;

    if IsFeatureImplemented(FEAT_PAuth2) && IsFeatureImplemented(FEAT_CONSTPACFIELD) then
        selbit = ptr<55>;
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(selbit);

    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
    else
        extfield = Replicate(selbit, 64);

    // 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>;
elseif mtX then
    if bottom_PAC_bit <= 55 then
        ext_ptr = (extfield<63:60> : ptr<59:56> :
                    extfield<55:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>);
    else
        ext_ptr = extfield<63:60> : ptr<59:56> : ptr<55:0>;
else
    ext_ptr = extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

if use_modifier2 then
    assert IsFeatureImplemented(FEAT_PAuth_LR);
    PAC = ComputePAC2(ext_ptr, modifier, modifier2, K<127:64>, K<63:0>, isgeneric);
else
    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>, isgeneric);

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 !mtX;
    assert bottom_PAC_bit <= 52;
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if IsFeatureImplemented(FEAT_EPAC) then
            PAC = 0x0000000000000000<63:0>;
        else
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

// 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>;
        // A compliant implementation of FEAT_MTE4 also implements FEAT_PAuth2
        assert !mtX;
else
    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> EOR PAC<55>;
    else
        bit55 = selbit;
    if tbi then
        if bottom_PAC_bit < 55 then
            result = (ptr<63:56> : bit55 :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
        else
            result = (ptr<63:56> : bit55 : ptr<54:0>);
    elseif mtX then
        if bottom_PAC_bit < 55 then
            result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : bit55 :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);
        else
            result = ((ptr<63:60> EOR PAC<63:60>) : ptr<59:56> : bit55 :
                      ptr<54:0>);
    else
        if bottom_PAC_bit < 55 then
            result = ((ptr<63:56> EOR PAC<63:56>) : bit55 :
                      (ptr<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>) :
                      ptr<bottom_PAC_bit-1:0>);

```

```

        else
            result = ((ptr<63:56> EOR PAC<63:56>) : bit55 :
                    ptr<54:0>);
    return result;

```

#### 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.

bits(64) AddPACDA(bits(64) x, bits(64) y)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDAKey_EL1, TRUE);

```

#### 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.

bits(64) AddPACDB(bits(64) x, bits(64) y)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APDBKey_EL1, TRUE);

```

## 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.

bits(64) AddPACGA(bits(64) x, bits(64) y)
    boolean TrapEL2;
    constant boolean isgeneric = TRUE;
    constant bits(128) APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;

    boolean TrapEL3;
    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;

    if TrapEL3 && EL3SDDUndefPriority() then
        UNDEFINED;
    elseif TrapEL2 then
        TrapPACUse(EL2);
    elseif TrapEL3 then
        if EL3SDDUndef() then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(x, y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0>, isgeneric)<63:32>:Zeros(32);
```

## 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.

bits(64) AddPACIA(bits(64) x, bits(64) y)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIAKey_EL1, FALSE);
```

## 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.

bits(64) AddPACIA2(bits(64) x, bits(64) y, bits(64) z)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIAKey_EL1, FALSE);
```

## 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.

bits(64) AddPACIB(bits(64) x, bits(64) y)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC(x, y, APIBKey_EL1, FALSE);
```

## 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.

bits(64) AddPACIB2(bits(64) x, bits(64) y, bits(64) z)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return AddPAC2(x, y, z, APIBKey_EL1, FALSE);
```

## Library pseudocode for aarch64/functions/pac/auth/AArch64.PACFailException

```
// AArch64.PACFailException()
// =====
// Generates a PAC Fail Exception

AArch64.PACFailException(bits(2) syndrome)
    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = 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);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, except, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, except, preferred_exception_return, vect_offset);
```

## 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.

bits(64) Auth(bits(64) ptr, bits(64) modifier, bits(128) K, boolean data, bit key_number,
    boolean is_combined)
    constant boolean use_modifier2 = FALSE;
    return Authenticate(ptr, modifier, Zeros(64), use_modifier2, K, data, key_number, is_combined);
```

## 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.

bits(64) Auth2(bits(64) ptr, bits(64) modifier1, bits(64) modifier2, bits(128) K,
    boolean data, bit key_number, boolean is_combined)
    constant boolean use_modifier2 = TRUE;
    return Authenticate(ptr, modifier1, modifier2, use_modifier2, K, data, key_number, is_combined);
```



```

// 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.

bits(64) Authenticate(bits(64) ptr, bits(64) modifier, bits(64) modifier2, boolean use_modifier2,
                      bits(128) K, boolean data, bit key_number, boolean is_combined)

    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;
    constant boolean isgeneric = FALSE;

    // Reconstruct the extension field used of adding the PAC to the pointer
    constant boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    constant boolean mtz = EffectiveMTX(ptr, !data, PSTATE.EL) == '1';
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
                                       TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
                                       AArch64.IASize(TCR_EL3.T0SZ) > 52);
    constant boolean is_VA_56bit = (TranslationRegime(PSTATE.EL) == Regime\_EL3 &&
                                    AArch64.IASize(TCR_EL3.T0SZ) == 56);

    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
    else
        extfield = Replicate(ptr<55>, 64);

    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>;
    elsif mtz 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>;
    else
        original_ptr = extfield<63:bottom_PAC_bit> : ptr<bottom_PAC_bit-1:0>;

    if use_modifier2 then
        assert IsFeatureImplemented(FEAT_PAuth_LR);
        PAC = ComputePAC2(original_ptr, modifier, modifier2, K<127:64>, K<63:0>, isgeneric);
    else
        PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>, isgeneric);
    // 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>;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result<55> = result<55> EOR PAC<55>;
            if (bottom_PAC_bit < 55) then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            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);
    elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
            result<54:bottom_PAC_bit> !=
            Replicate(result<55>, (55-bottom_PAC_bit))) then
        error_code = (if data then '1' else '0'):key_number;
        AArch64.PACFailException(error_code);
elsif mtx then
    assert IsFeatureImplemented(FEAT_PAuth2);
    result = ptr;
    if EL3_using_lva3 && !is_VA_56bit then
        result<55> = result<55> EOR PAC<55>;
    if (bottom_PAC_bit < 55) then
        result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
    result<63:60> = result<63:60> EOR 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);
        elsif (!EL3_using_lva3 && (bottom_PAC_bit < 55) &&
            (((result<54:bottom_PAC_bit> !=
                Replicate(result<55>, (55-bottom_PAC_bit)))) ||
            (result<63:60> != Replicate(result<55>, 4)))) then
            error_code = (if data then '1' else '0'):key_number;
            AArch64.PACFailException(error_code);
    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>;
        else
            result = ptr;
            if EL3_using_lva3 && !is_VA_56bit then
                result<55> = result<55> EOR PAC<55>;
            if bottom_PAC_bit < 55 then
                result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            result<63:56> = result<63:56> EOR 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);
                elsif (!EL3_using_lva3 &&
                    result<63:bottom_PAC_bit> !=
                    Replicate(result<55>, (64-bottom_PAC_bit))) then
                    error_code = (if data then '1' else '0'):key_number;
                    AArch64.PACFailException(error_code);
        return result;

```

## 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().

bits(64) AuthDA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    if !IsAPDAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDAKey_EL1, TRUE, '0', is_combined);

```



## 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().

bits(64) AuthDB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    if !IsAPDBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APDBKey_EL1, TRUE, '1', is_combined);
```

## 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().

bits(64) AuthIA(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    if !IsAPIAKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIAKey_EL1, FALSE, '0', is_combined);
```

## 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().

bits(64) AuthIA2(bits(64) x, bits(64) y, bits(64) z, boolean is_combined)
    constant bits(128) APIAKey_EL1 = 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);
```

## 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().

bits(64) AuthIB(bits(64) x, bits(64) y, boolean is_combined)
    constant bits(128) APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    if !IsAPIBKeyEnabled() then
        return x;
    else
        return Auth(x, y, APIBKey_EL1, FALSE, '1', is_combined);
```

## 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().

bits(64) AuthIB2(bits(64) x, bits(64) y, bits(64) z, boolean is_combined)
    constant bits(128) APIBKey_EL1 = 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);
```

## 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

bits(6) AArch64.PACEffectiveTxSZ(Regime regime, S1TTWParams walkparams)
    constant integer slmaxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    constant integer slmintxsz = AArch64.S1MinTxSZ(regime, walkparams.dl28,
                                                    walkparams.ds, walkparams.tgx);

    if AArch64.S1TxSZFaults(regime, walkparams) then
        if ConstrainUnpredictable(Unpredictable RESTnSZ) == Constraint_FORCE then
            if UInt(walkparams.txsz) < slmintxsz then
                return slmintxsz<5:0>;
            if UInt(walkparams.txsz) > slmaxtxsz then
                return slmaxtxsz<5:0>;
        elseif UInt(walkparams.txsz) < slmintxsz then
            return slmintxsz<5:0>;
        elseif UInt(walkparams.txsz) > slmaxtxsz then
            return slmaxtxsz<5:0>;

    return walkparams.txsz;
```

## Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

AddressSize CalculateBottomPACBit(bit top_bit)
    Regime regime;
    S1TTWParams walkparams;
    AddressSize bottom_PAC_bit;

    regime = TranslationRegime(PSTATE.EL);
    ss = CurrentSecurityState();
    walkparams = AArch64.GetS1TTWParams(regime, ss, Replicate(top_bit, 64));
    bottom_PAC_bit = 64 - UInt(AArch64.PACEffectiveTxSZ(regime, walkparams));

    return bottom_PAC_bit;
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC

```
// ComputePAC()
// =====

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1,
                    boolean isgeneric)
    if UsePACIMP(isgeneric) then
        return ComputePACIMPDEF(data, modifier, key0, key1);
    if UsePACQARMA3(isgeneric) then
        constant boolean isqarma3 = TRUE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    if UsePACQARMA5(isgeneric) then
        constant boolean isqarma3 = FALSE;
        return ComputePACQARMA(data, modifier, key0, key1, isqarma3);
    Unreachable();
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2

```
// ComputePAC2()
// =====

bits(64) ComputePAC2(bits(64) data, bits(64) modifier1, bits(64) modifier2,
                    bits(64) key0, bits(64) key1, boolean isgeneric)
    if UsePACIMP(isgeneric) then
        return ComputePAC2IMPDEF(data, modifier1, modifier2, key0, key1);
    if UsePACQARMA3(isgeneric) then
        constant boolean isqarma3 = TRUE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    if UsePACQARMA5(isgeneric) then
        constant boolean isqarma3 = FALSE;
        return ComputePAC2QARMA(data, modifier1, modifier2, key0, key1, isqarma3);
    Unreachable();
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2IMPDEF

```
// ComputePAC2IMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePAC2IMPDEF(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                           bits(64) key1);
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePAC2QARMA

```
// ComputePAC2QARMA()
// =====

bits(64) ComputePAC2QARMA(bits(64) data, bits(64) modifier1, bits(64) modifier2, bits(64) key0,
                          bits(64) key1, boolean isqarma3)
    constant bits(64) concat_modifiers = modifier2<36:5>:modifier1<35:4>;
    return ComputePACQARMA(data, concat_modifiers, key0, key1, isqarma3);
```

## Library pseudocode for aarch64/functions/pac/computepac/ComputePACIMPDEF

```
// ComputePACIMPDEF()
// =====
// Compute IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC calculation.

bits(64) ComputePACIMPDEF(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1);
```



```

// ComputePACQARMA()
// =====
// Compute QARMA3 or QARMA5 cryptographic algorithm for PAC calculation

bits(64) ComputePACQARMA(bits(64) data, bits(64) modifier, bits(64) key0,
                        bits(64) key1, boolean isqarma3)

bits(64) workingval;
bits(64) runningmod;
bits(64) roundkey;
bits(64) modk0;
constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

integer iterations;
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>;

modk0 = key0<0>:key0<63:2>:(key0<63> EOR key0<1>);
runningmod = modifier;
workingval = data EOR key0;

for i = 0 to iterations
    roundkey = key1 EOR runningmod;
    workingval = workingval EOR roundkey;
    workingval = workingval EOR RC[i];
    if i > 0 then
        workingval = PACCellShuffle(workingval);
        workingval = PACMult(workingval);
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACSub(workingval);
    runningmod = TweakShuffle(runningmod<63:0>);
roundkey = modk0 EOR runningmod;
workingval = workingval EOR roundkey;
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACSub(workingval);
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
workingval = key1 EOR workingval;
workingval = PACCellInvShuffle(workingval);
if isqarma3 then
    workingval = PACSub1(workingval);
else
    workingval = PACInvSub(workingval);
workingval = PACMult(workingval);
workingval = PACCellInvShuffle(workingval);
workingval = workingval EOR key0;
workingval = workingval EOR runningmod;
for i = 0 to iterations
    if isqarma3 then
        workingval = PACSub1(workingval);
    else
        workingval = PACInvSub(workingval);
    if i < iterations then
        workingval = PACMult(workingval);
        workingval = PACCellInvShuffle(workingval);
    runningmod = TweakInvShuffle(runningmod<63:0>);
    roundkey = key1 EOR runningmod;

```

```

        workingval = workingval EOR RC[iterations-i];
        workingval = workingval EOR roundkey;
        workingval = workingval EOR Alpha;
    workingval = workingval EOR modk0;

    return workingval;

```

### Library pseudocode for aarch64/functions/pac/computepac/PACCellInvShuffle

```

// PACCellInvShuffle()
// =====

bits(64) PACCellInvShuffle(bits(64) indata)
    bits(64) outdata;
    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;

```

### Library pseudocode for aarch64/functions/pac/computepac/PACCellShuffle

```

// PACCellShuffle()
// =====

bits(64) PACCellShuffle(bits(64) indata)
    bits(64) outdata;
    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;

```

## Library pseudocode for aarch64/functions/pac/computepac/PACInvSub

```
// PACInvSub()
// =====

bits(64) PACInvSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Elem[Tinput, i, 4] of
        when '0000' Elem[Toutput, i, 4] = '0101';
        when '0001' Elem[Toutput, i, 4] = '1110';
        when '0010' Elem[Toutput, i, 4] = '1101';
        when '0011' Elem[Toutput, i, 4] = '1000';
        when '0100' Elem[Toutput, i, 4] = '1010';
        when '0101' Elem[Toutput, i, 4] = '1011';
        when '0110' Elem[Toutput, i, 4] = '0001';
        when '0111' Elem[Toutput, i, 4] = '1001';
        when '1000' Elem[Toutput, i, 4] = '0010';
        when '1001' Elem[Toutput, i, 4] = '0110';
        when '1010' Elem[Toutput, i, 4] = '1111';
        when '1011' Elem[Toutput, i, 4] = '0000';
        when '1100' Elem[Toutput, i, 4] = '0100';
        when '1101' Elem[Toutput, i, 4] = '1100';
        when '1110' Elem[Toutput, i, 4] = '0111';
        when '1111' Elem[Toutput, i, 4] = '0011';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACMult

```
// PACMult()
// =====

bits(64) PACMult(bits(64) Sinput)
bits(4) t0;
bits(4) t1;
bits(4) t2;
bits(4) t3;
bits(64) Soutput;

for i = 0 to 3
    t0<3:0> = ROL(Elem[Sinput, (i+8), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 2);
    t0<3:0> = t0<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
    t1<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+4), 4], 1);
    t1<3:0> = t1<3:0> EOR ROL(Elem[Sinput, i, 4], 2);
    t2<3:0> = ROL(Elem[Sinput, (i+12), 4], 2) EOR ROL(Elem[Sinput, (i+8), 4], 1);
    t2<3:0> = t2<3:0> EOR ROL(Elem[Sinput, i, 4], 1);
    t3<3:0> = ROL(Elem[Sinput, (i+12), 4], 1) EOR ROL(Elem[Sinput, (i+8), 4], 2);
    t3<3:0> = t3<3:0> EOR ROL(Elem[Sinput, (i+4), 4], 1);
    Elem[Soutput, i, 4] = t3<3:0>;
    Elem[Soutput, (i+4), 4] = t2<3:0>;
    Elem[Soutput, (i+8), 4] = t1<3:0>;
    Elem[Soutput, (i+12), 4] = t0<3:0>;
return Soutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PACSub

```
// PACSub()
// =====

bits(64) PACSub(bits(64) Tinput)
// This is a 4-bit substitution from the PRINCE-family cipher
bits(64) Toutput;
for i = 0 to 15
    case Elem[Tinput, i, 4] of
        when '0000' Elem[Toutput, i, 4] = '1011';
        when '0001' Elem[Toutput, i, 4] = '0110';
        when '0010' Elem[Toutput, i, 4] = '1000';
        when '0011' Elem[Toutput, i, 4] = '1111';
        when '0100' Elem[Toutput, i, 4] = '1100';
        when '0101' Elem[Toutput, i, 4] = '0000';
        when '0110' Elem[Toutput, i, 4] = '1001';
        when '0111' Elem[Toutput, i, 4] = '1110';
        when '1000' Elem[Toutput, i, 4] = '0011';
        when '1001' Elem[Toutput, i, 4] = '0111';
        when '1010' Elem[Toutput, i, 4] = '0100';
        when '1011' Elem[Toutput, i, 4] = '0101';
        when '1100' Elem[Toutput, i, 4] = '1101';
        when '1101' Elem[Toutput, i, 4] = '0010';
        when '1110' Elem[Toutput, i, 4] = '0001';
        when '1111' Elem[Toutput, i, 4] = '1010';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/PacSub1

```
// PacSub1()
// =====

bits(64) PACSub1(bits(64) Tinput)
// This is a 4-bit substitution from Qarma signal
bits(64) Toutput;
for i = 0 to 15
    case Elem[Tinput, i, 4] of
        when '0000' Elem[Toutput, i, 4] = '1010';
        when '0001' Elem[Toutput, i, 4] = '1101';
        when '0010' Elem[Toutput, i, 4] = '1110';
        when '0011' Elem[Toutput, i, 4] = '0110';
        when '0100' Elem[Toutput, i, 4] = '1111';
        when '0101' Elem[Toutput, i, 4] = '0111';
        when '0110' Elem[Toutput, i, 4] = '0011';
        when '0111' Elem[Toutput, i, 4] = '0101';
        when '1000' Elem[Toutput, i, 4] = '1001';
        when '1001' Elem[Toutput, i, 4] = '1000';
        when '1010' Elem[Toutput, i, 4] = '0000';
        when '1011' Elem[Toutput, i, 4] = '1100';
        when '1100' Elem[Toutput, i, 4] = '1011';
        when '1101' Elem[Toutput, i, 4] = '0001';
        when '1110' Elem[Toutput, i, 4] = '0010';
        when '1111' Elem[Toutput, i, 4] = '0100';
return Toutput;
```

## Library pseudocode for aarch64/functions/pac/computepac/RC

```
// RC[]
// ====

array bits(64) RC[0..4];
```



## Library pseudocode for aarch64/functions/pac/computepac/TweakCellInvRot

```
// TweakCellInvRot()
// =====

bits(4) TweakCellInvRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<2>;
    outcell<2> = incell<1>;
    outcell<1> = incell<0>;
    outcell<0> = incell<0> EOR incell<3>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakCellRot

```
// TweakCellRot()
// =====

bits(4) TweakCellRot(bits(4) incell)
    bits(4) outcell;
    outcell<3> = incell<0> EOR incell<1>;
    outcell<2> = incell<3>;
    outcell<1> = incell<2>;
    outcell<0> = incell<1>;
    return outcell;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakInvShuffle

```
// TweakInvShuffle()
// =====

bits(64) TweakInvShuffle(bits(64) indata)
    bits(64) outdata;
    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;
```

## Library pseudocode for aarch64/functions/pac/computepac/TweakShuffle

```
// TweakShuffle()
// =====

bits(64) TweakShuffle(bits(64) indata)
    bits(64) outdata;
    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;
```

## Library pseudocode for aarch64/functions/pac/computepac/UsePACIMP

```
// UsePACIMP()
// =====
// Checks whether IMPLEMENTATION DEFINED cryptographic algorithm to be used for PAC
// calculation.

boolean UsePACIMP(boolean isgeneric)
    return if isgeneric then HavePACIMPGeneric() else HavePACIMPAddr();
```

## Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA3

```
// UsePACQARMA3()
// =====
// Checks whether QARMA3 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA3(boolean isgeneric)
    return if isgeneric then HavePACQARMA3Generic() else HavePACQARMA3Addr();
```

## Library pseudocode for aarch64/functions/pac/computepac/UsePACQARMA5

```
// UsePACQARMA5()
// =====
// Checks whether QARMA5 cryptographic algorithm to be used for PAC calculation.

boolean UsePACQARMA5(boolean isgeneric)
    return if isgeneric then HavePACQARMA5Generic() else HavePACQARMA5Addr();
```

## Library pseudocode for aarch64/functions/pac/pac/HavePACIMPAddr

```
// HavePACIMPAddr()
// =====
// Returns TRUE if support for PAC IMP for address authentication is implemented,
// FALSE otherwise.

boolean HavePACIMPAddr()
    return IsFeatureImplemented(FEAT_PACIMP);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACIMPGeneric

```
// HavePACIMPGeneric()
// =====
// Returns TRUE if support for PAC IMP for generic authentication is implemented,
// FALSE otherwise.

boolean HavePACIMPGeneric()
    return IsFeatureImplemented(FEAT_PACIMP);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Addr

```
// HavePACQARMA3Addr()
// =====
// Returns TRUE if support for PAC QARMA3 for address authentication is implemented,
// FALSE otherwise.

boolean HavePACQARMA3Addr()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3Generic

```
// HavePACQARMA3Generic()
// =====
// Returns TRUE if support for PAC QARMA3 for generic authentication is implemented,
// FALSE otherwise.

boolean HavePACQARMA3Generic()
    return IsFeatureImplemented(FEAT_PACQARMA3);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Addr

```
// HavePACQARMA5Addr()
// =====
// Returns TRUE if support for PAC QARMA5 for address authentication is implemented,
// FALSE otherwise.

boolean HavePACQARMA5Addr()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

### Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5Generic

```
// HavePACQARMA5Generic()
// =====
// Returns TRUE if support for PAC QARMA5 for generic authentication is implemented,
// FALSE otherwise.

boolean HavePACQARMA5Generic()
    return IsFeatureImplemented(FEAT_PACQARMA5);
```

## 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.

boolean IsAPDAKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime\(\) == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnDA else SCTLR_EL2.EnDA;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0' && !IsInHost\(\);
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnDA;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    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);
    else
        return TRUE;
```

## 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.

boolean IsAPDBKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime\(\) == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0' && !IsInHost\(\);
            TrapEL3 = HaveEL\(EL3\) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnDB;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0';
            TrapEL3 = HaveEL\(EL3\) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL\(EL3\) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnDB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    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\(EL3\);
    else
        return TRUE;
```

## 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.

boolean IsAPIAKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0' && !IsInHost();
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    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);
    else
        return TRUE;
```

## 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.

boolean IsAPIBKeyEnabled()
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;

    case PSTATE.EL of
        when EL0
            constant boolean IsEL1Regime = S1TranslationRegime\(\) == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0' && !IsInHost\(\);
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTLR_EL1.EnIB;
            TrapEL2 = EL2Enabled\(\) && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTLR_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTLR_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    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);
    else
        return TRUE;
```

## Library pseudocode for aarch64/functions/pac/pac/IsPACMEnabled

```
// IsPACMEnabled()
// =====
// Returns TRUE if the effects of the PACM instruction are enabled, otherwise FALSE.

boolean IsPACMEnabled()
    assert IsFeatureImplemented(FEAT_PAuth) && IsFeatureImplemented(FEAT_PAuth_LR);

    if IsTrivialPACMImplementation\(\) then
        return FALSE;

    boolean enabled;

    // 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;

    // Otherwise, the SCTLR2_ELx bit determines the behavior.
    if enabled then
        bit enpacm_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';
        enabled = enpacm_bit == '1';

    return enabled;
```

## Library pseudocode for aarch64/functions/pac/pac/IsTrivialPACMImplementation

```
// IsTrivialPACMImplementation()
// =====
// Returns TRUE if the PE has a trivial implementation of PACM.

boolean IsTrivialPACMImplementation()
    return (IsFeatureImplemented(FEAT_PACIMP) &&
        boolean IMPLEMENTATION_DEFINED "Trivial PSTATE.PACM implementation");
```

## Library pseudocode for aarch64/functions/pac/pac/PtrHasUpperAndLowerAddRanges

```
// PtrHasUpperAndLowerAddRanges()
// =====
// Returns TRUE if the pointer has upper and lower address ranges, FALSE otherwise.

boolean PtrHasUpperAndLowerAddRanges()
    regime = TranslationRegime(PSTATE.EL);
    return HasUnprivileged(regime);
```



## 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.

bits(64) Strip(bits(64) A, boolean data)
    bits(64) original_ptr;
    bits(64) extfield;
    constant boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    constant boolean mtX = EffectiveMTX(A, !data, PSTATE.EL) == '1';
    constant AddressSize bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    constant boolean EL3_using_lva3 = (IsFeatureImplemented(FEAT_LVA3) &&
        TranslationRegime(PSTATE.EL) == Regime EL3 &&
        AArch64.IASize(TCR_EL3.T0SZ) > 52);

    if EL3_using_lva3 then
        extfield = Replicate('0', 64);
    else
        extfield = Replicate(A<55>, 64);

    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>;
    elsif 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>;
    else
        original_ptr = extfield<63:bottom_PAC_bit> : A<bottom_PAC_bit-1:0>;

    return original_ptr;
```

## Library pseudocode for aarch64/functions/pac/trappacuse/TrapPACUse

```
// TrapPACUse()
// =====
// Used for the trapping of the pointer authentication functions by higher exception
// levels.

TrapPACUse(bits(2) target_el)
    assert HaveEL(target_el) && target_el != EL0 && UInt(target_el) >= UInt(PSTATE.EL);

    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    ExceptionRecord except;
    vect_offset = 0;
    except = ExceptionSyndrome(Exception\_PACTrap);
    AArch64.TakeException(target_el, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/predictionrestrict/AArch64.RestrictPrediction

```
// AArch64.RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = 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();

    constant bit ns  = val<26>;
    constant bit nse = val<27>;
    ss = 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();
    if !IsFeatureImplemented(FEAT_RME) && target_el == EL3 && ss != SS_Secure then
        ExecuteAsNOP();

    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[];

            elseif (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;
        else
            c.is_vmid_valid = FALSE;

    if PSTATE.EL == EL0 then
        c.is_asid_valid = TRUE;
        c.all_asid      = FALSE;
        c.asid          = ASID[];

    elseif 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;

    c.restriction = restriction;
    RESTRICT_PREDICTIONS(c);
```

## Library pseudocode for aarch64/functions/prefetch/Prefetch

```
// Prefetch()
// =====

// Decode and execute the prefetch hint on ADDRESS specified by PRFOP

Prefetch(bits(64) address, bits(5) prfop)
    PrefetchHint hint;
    integer target;
    boolean stream;

    case prfop<4:3> of
        when '00' hint = Prefetch\_READ;           // PLD: prefetch for load
        when '01' hint = Prefetch\_EXEC;           // PLI: preload instructions
        when '10' hint = Prefetch\_WRITE;          // PST: prepare for store
        when '11' return;                          // unallocated hint
    target = UInt(prfop<2:1>);                      // target cache level
    stream = (prfop<0> != '0');                    // streaming (non-temporal)
    Hint\_Prefetch(address, hint, target, stream);
    return;
```

## Library pseudocode for aarch64/functions/pstatefield/PSTATEField

```
// PSTATEField
// =====
// MSR (immediate) instruction destinations.

enumeration PSTATEField {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.

(boolean, bits(2)) AArch64.DelegatedSErrorTarget()
    assert IsFeatureImplemented(FEAT_E3DSE);
    if Halted\(\) || PSTATE.EL == EL3 then
        return (TRUE, bits(2) UNKNOWN);

    constant bit effective_amo = EffectiveHCR\_AMO\(\);
    constant bit effective_tge = EffectiveTGE\(\);
    constant bit effective_nmea = EffectiveNMEA\(\);

    // The exception is masked by software.
    boolean masked;
    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();

    // When FEAT_DoubleFault or FEAT_DoubleFault2 is implemented, the mask might be overridden.
    masked = (masked && effective_nmea == '0');

    // The exception might be disabled debug in the Security state indicated by
    // SCR_EL3.{NS, NSE} by external debug.
    boolean intdis = ExternalDebugInterruptsDisabled\(EL1\);

    bits(2) target_el = bits(2) UNKNOWN;
    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;

    else
        assert (PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)));
        if !masked then target_el = EL1;

    // External debug might disable the delegated exception for the target Exception level.
    if ExternalDebugInterruptsDisabled\(target\_el\) then
        masked = TRUE;
        target_el = bits(2) UNKNOWN;

    return (masked, target_el);
```

## 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

AArch64.ESBOperation()
    bits(2) target_el;
    boolean masked;

    (masked, target_el) = PhysicalErrorTarget();

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalErrorPending() then
        // This function might be called for an interprocessing case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32\(S1TranslationRegime\)() then
            bits(32) syndrome = Zeros(32);
            syndrome<31> = '1'; // A
            syndrome<15:0> = AArch32.PhysicalErrorSyndrome();
            DISR = syndrome;
        else
            implicit_esb = FALSE;
            bits(64) syndrome = Zeros(64);
            syndrome<31> = '1'; // A
            syndrome<24:0> = AArch64.PhysicalErrorSyndrome(implicit_esb);
            DISR_EL1 = syndrome;
            ClearPendingPhysicalError(); // Set ISR_EL1.A to 0

    return;
```

## Library pseudocode for aarch64/functions/ras/AArch64.EncodeAsyncErrorSyndrome

```
// AArch64.EncodeAsyncErrorSyndrome()
// =====
// Return the encoding for specified ErrorState for an SError exception taken
// to AArch64 state.

bits(3) AArch64.EncodeAsyncErrorSyndrome(ErrorState errorstate)
    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();
```

## 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.

bits(2) AArch64.EncodeSyncErrorSyndrome(ErrorState errorstate)
    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();
```

## Library pseudocode for aarch64/functions/ras/AArch64.PhysicalErrorSyndrome

```
// AArch64.PhysicalErrorSyndrome()
// =====
// Generate SError syndrome.

bits(25) AArch64.PhysicalErrorSyndrome(boolean implicit_esb)
    bits(25) syndrome = Zeros(25);

    if ReportErrorAsUncategorized() then
        syndrome = Zeros(25);
    elseif ReportErrorAsIMPDEF() then
        syndrome<24> = '1'; // IDS
        syndrome<23:0> = bits(24) IMPLEMENTATION_DEFINED "IMPDEF ErrorState";
    else
        constant FaultRecord fault = GetPendingPhysicalError();
        constant ErrorState errorstate = PEErrorState(fault);
        syndrome<24> = '0'; // IDS
        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

    return syndrome;
```

## Library pseudocode for aarch64/functions/ras/AArch64.dESBOperation

```
// AArch64.dESBOperation()
// =====
// Perform the AArch64 ESB operation for a pending delegated SError exception.

AArch64.dESBOperation()
    assert (IsFeatureImplemented(FEAT_E3DSE) && !ELUsingAArch32(EL3) && PSTATE.EL != EL3);
    // When FEAT_E3DSE is implemented, SCR_EL3.DSE might inject a delegated SError exception.
    boolean dsei_pending, dsei_masked;
    dsei_pending = SCR_EL3.<EnDSE,DSE> == '11';
    (dsei_masked, -) = AArch64.DelegatedSErrorTarget();
    if dsei_pending && dsei_masked then
        bits(64) target = Zeros(64);
        target<31> = '1'; // A
        target<24:0> = VESR_EL3<24:0>;
        VDISR_EL3 = target;
        ClearPendingDelegatedSError();
    return;
```

## 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.

AArch64.vESBOperation()
    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.
    vsei_pending = (IsVirtualSErrorPending() && EffectiveTGE() == '0' &&
        (EffectiveHCR_AMO() == '1' || EffectiveHCRX_EL2_TMEA() == '1'));
    vsei_masked = 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
            bits(32) target = Zeros(32);
            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
            VDISR = target;
        else
            bits(64) target = Zeros(64);
            target<31> = '1'; // A
            target<24:0> = VSESR_EL2<24:0>;
            VDISR_EL2 = target;
            ClearPendingVirtualSError();
    elseif IsFeatureImplemented(FEAT_E3DSE) then
        AArch64.dESBOperation();

    return;
```

## Library pseudocode for aarch64/functions/ras/FirstRecordOfNode

```
// FirstRecordOfNode()
// =====
// Return the first record in the node that contains the record n.

integer FirstRecordOfNode(integer n)
    for q = n downto 0
        if IsFirstRecordOfNode(q) then return q;
    Unreachable();
```

## 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.

boolean IsCommonFaultInjectionImplemented(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsCountableErrorsRecorded

```
// IsCountableErrorsRecorded()
// =====
// Check whether Error record n records countable errors.

boolean IsCountableErrorsRecorded(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsErrorAddressIncluded

```
// IsErrorAddressIncluded()
// =====
// Check whether Error record n includes an address associated with an error.

boolean IsErrorAddressIncluded(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsErrorRecordImplemented

```
// IsErrorRecordImplemented()
// =====
// Is the error record n implemented

boolean IsErrorRecordImplemented(integer n);
```

### Library pseudocode for aarch64/functions/ras/IsFirstRecordOfNode

```
// IsFirstRecordOfNode()
// =====
// Check if the record q is the first error record in its node.

boolean IsFirstRecordOfNode(integer q);
```

### Library pseudocode for aarch64/functions/ras/IsSPMUCounterImplemented

```
// IsSPMUCounterImplemented()
// =====
// Does the System PMU s implement the counter n.

boolean IsSPMUCounterImplemented(integer s, integer n);
```

### Library pseudocode for aarch64/functions/rcw/ProtectionEnabled

```
// ProtectionEnabled()
// =====
// Returns TRUE if the ProtectedBit is
// enabled in the current Exception level.

boolean ProtectionEnabled(bits(2) el)
    assert HaveEL(el);
    regime = 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';
    else
        return TRUE;
    return FALSE;
```



### Library pseudocode for aarch64/functions/rcw/RCW128\_PROTECTED\_BIT

```
constant integer RCW128_PROTECTED_BIT = 114;
```

### Library pseudocode for aarch64/functions/rcw/RCW64\_PROTECTED\_BIT

```
constant integer RCW64_PROTECTED_BIT = 52;
```



```

// RCWCheck()
// =====
// Returns nzcw 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

bits(4) RCWCheck(bits(N) old, bits(N) new, boolean soft)
    assert N IN {64,128};
    constant integer protectedbit = if N == 128 then RCW128\_PROTECTED\_BIT else RCW64\_PROTECTED\_BIT;
    boolean rcw_fail = FALSE;
    boolean rcws_fail = FALSE;
    boolean rcw_state_fail = FALSE;
    boolean rcws_state_fail = FALSE;
    boolean rcw_mask_fail = FALSE;
    boolean rcws_mask_fail = FALSE;

    //Effective RCWMask calculation
    bits(N) rcwmask = RCWMASK_EL1<N-1:0>;
    if N == 64 then
        rcwmask<49:18> = Replicate(rcwmask<17>,32);
        rcwmask<0> = '0';
    else
        rcwmask<55:17> = Replicate(rcwmask<16>,39);
        rcwmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);

    //Effective RCWSMask calculation
    bits(N) rcwsoftmask = RCWSMASK_EL1<N-1:0>;
    if N == 64 then
        rcwsoftmask<49:18> = Replicate(rcwsoftmask<17>,32);
        rcwsoftmask<0> = '0';
        if(ProtectionEnabled(PSTATE.EL)) then
            rcwsoftmask<52> = '0';
    else
        rcwsoftmask<55:17> = Replicate(rcwsoftmask<16>,39);
        rcwsoftmask<126:125,120:119,107:101,90:56,1:0> = Zeros(48);
        rcwsoftmask<114> = '0';

    //RCW Checks
    //State Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit> == '1' then
            rcw_state_fail = new<protectedbit,0> != old<protectedbit,0>;
        elsif old<protectedbit> == '0' then
            rcw_state_fail = new<protectedbit> != old<protectedbit>;

    //Mask Check
    if (ProtectionEnabled(PSTATE.EL)) then
        if old<protectedbit,0> == '11' then
            rcw_mask_fail = IsZero((new EOR old) AND NOT(rcwmask));

    //RCWS Checks
    if soft then
        //State Check
        if old<0> == '1' then
            rcws_state_fail = new<0> != old<0>;
        elsif (!ProtectionEnabled(PSTATE.EL) ||
            (ProtectionEnabled(PSTATE.EL) && old<protectedbit> == '0')) then
            rcws_state_fail = new<0> != old<0>;
        //Mask Check
        if old<0> == '1' then
            rcws_mask_fail = IsZero((new EOR old) AND NOT(rcwsoftmask));

    rcw_fail = rcw_state_fail || rcw_mask_fail ;
    rcws_fail = rcws_state_fail || rcws_mask_fail;

    constant bit n = '0';
    constant bit z = if rcw_fail then '1' else '0';
    constant bit c = if rcws_fail then '0' else '1';
    constant bit v = '0';
    return n:z:c:v;

```

## 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.

bits(esize) FPReduce(ReduceOp op, bits(N) input, integer esize, FPCR_Type fpcr)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;

    if N == esize then
        return input<esize-1:0>;

    hi = FPReduce(op, input<N-1:half>, esize, fpcr);
    lo = FPReduce(op, input<half-1:0>, esize, fpcr);
    case op of
        when ReduceOp_FMINNUM
            result = FPMINNum(lo, hi, fpcr);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, fpcr);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, fpcr);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, fpcr);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, fpcr);

    return result;
```

## 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.

bits(esize) IntReduce(ReduceOp op, bits(N) input, integer esize)
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;
    constant integer half = N DIV 2;
    if N == esize then
        return input<esize-1:0>;

    hi = IntReduce(op, input<N-1:half>, esize);
    lo = IntReduce(op, input<half-1:0>, esize);
    case op of
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

## Library pseudocode for aarch64/functions/reduceop/ReduceOp

```
// ReduceOp
// =====
// Vector reduce instruction types.

enumeration ReduceOp {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.

AArch64.MaybeZeroRegisterUppers()
    assert UsingAArch32();          // Always called from AArch32 state before entering AArch64 state

    integer first;
    integer last;
    boolean include_R15;
    if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then
        first = 0; last = 14; include_R15 = FALSE;
    elsif PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !ELUsingAArch32(EL2) then
        first = 0; last = 30; include_R15 = FALSE;
    else
        first = 0; last = 30; include_R15 = TRUE;

    for n = first to last
        if (n != 15 || include_R15) && ConstrainUnpredictableBool(Unpredictable\_ZEROUPPER) then
            _R[n]<63:32> = Zeros(32);

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i, 64] = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i, 128] = bits(128) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSpecialRegisters

```
// AArch64.ResetSpecialRegisters()
// =====

AArch64.ResetSpecialRegisters()

    // AArch64 special registers
    SP_EL0 = bits(64) UNKNOWN;
    SP_EL1 = bits(64) UNKNOWN;

    SPSR_EL1 = bits(64) UNKNOWN;
    ELR_EL1 = bits(64) UNKNOWN;
    if HaveEL(EL2) then
        SP_EL2 = bits(64) UNKNOWN;
        SPSR_EL2 = bits(64) UNKNOWN;
        ELR_EL2 = bits(64) UNKNOWN;

    // AArch32 special registers that are not architecturally mapped to AArch64 registers
    if HaveAArch32EL(EL1) then
        SPSR_fiq<31:0> = bits(32) UNKNOWN;
        SPSR_irq<31:0> = bits(32) UNKNOWN;
        SPSR_abt<31:0> = bits(32) UNKNOWN;
        SPSR_und<31:0> = bits(32) UNKNOWN;

    // External debug special registers
    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;

    return;
```

## Library pseudocode for aarch64/functions/registers/AArch64.ResetSystemRegisters

```
// AArch64.ResetSystemRegisters()
// =====

AArch64.ResetSystemRegisters(boolean cold_reset);
```

## Library pseudocode for aarch64/functions/registers/ESize

```
// SIMD and Floating-point registers
// ++++++

// ESize
// =====

type ESize = integer;
```

## Library pseudocode for aarch64/functions/registers/PC64

```
// Program counter
// ++++++

// PC64 - non-assignment form
// =====
// Read program counter.

bits(64) PC64
    return _PC;
```

## Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write a 32-bit or 64-bit value to the current stack pointer.

SP[integer width] = bits(width) value
    assert width IN {64, 32};
    if PSTATE.SP == '0' then
        SP_EL0 = ZeroExtend(value, 64);
    else
        case PSTATE.EL of
            when EL0   SP_EL0 = ZeroExtend(value, 64);
            when EL1   SP_EL1 = ZeroExtend(value, 64);
            when EL2   SP_EL2 = ZeroExtend(value, 64);
            when EL3   SP_EL3 = ZeroExtend(value, 64);
        return;

// SP[] - non-assignment form
// =====
// Read the least-significant 32 or 64 bits from the current stack pointer.

bits(width) SP[integer width]
    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>;
```

## Library pseudocode for aarch64/functions/registers/SPMCFGR\_EL1

```
// SPMCFGR_EL1[] - non-assignment form
// =====
// Read the current configuration of System Performance monitor for
// System PMU 's'.

bits(64) SPMCFGR_EL1[integer s];
```

## Library pseudocode for aarch64/functions/registers/SPMCGCR\_EL1

```
// SPMCGCR_EL1[] - non-assignment form
// =====
// Read counter group 'n' configuration for System PMU 's'.

bits(64) SPMCGCR_EL1[integer s, integer n];
```

## Library pseudocode for aarch64/functions/registers/SPMCNTENCLR\_EL0

```
// SPMCNTENCLR_EL0[] - non-assignment form
// =====
// Read the current mapping of disabled event counters for an 's'.

bits(64) SPMCNTENCLR_EL0[integer s];

// SPMCNTENCLR_EL0[] - assignment form
// =====
// Disable event counters for System PMU 's'.

SPMCNTENCLR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMCNTENSET\_EL0

```
// SPMCNTENSET_EL0[] - non-assignment form
// =====
// Read the current mapping for enabled event counters of System PMU 's'.

bits(64) SPMCNTENSET_EL0[integer s];

// SPMCNTENSET_EL0[] - assignment form
// =====
// Enable event counters of System PMU 's'.

SPMCNTENSET_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMCR\_EL0

```
// SPMCR_EL0[] - non-assignment form
// =====
// Read the control register for System PMU 's'.

bits(64) SPMCR_EL0[integer s];

// SPMCR_EL0[] - assignment form
// =====
// Write to the control register for System PMU 's'.

SPMCR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMDEVAFF\_EL1

```
// SPMDEVAFF_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVAFF_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMDEVARCH\_EL1

```
// SPMDEVARCH_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMDEVARCH_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMEVCNTR\_EL0

```
// SPMEVCNTR_EL0[] - non-assignment form
// =====
// Read a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

bits(64) SPMEVCNTR_EL0[integer s, integer n];

// SPMEVCNTR_EL0[] - assignment form
// =====
// Write to a System PMU Event Counter register for counter 'n' of a given
// System PMU 's'.

SPMEVCNTR_EL0[integer s, integer n] = bits(64) value;
```



### Library pseudocode for aarch64/functions/registers/SPMEVFILT2R\_EL0

```
// SPMEVFILT2R_EL0[] - non-assignment form
// =====
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILT2R_EL0[integer s, integer n];

// SPMEVFILT2R_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILT2R_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMEVFILTR\_EL0

```
// SPMEVFILTR_EL0[] - non-assignment form
// =====
// Read the additional event selection controls for
// counter 'n' of a given System PMU 's'.

bits(64) SPMEVFILTR_EL0[integer s, integer n];

// SPMEVFILTR_EL0[] - assignment form
// =====
// Configure the additional event selection controls for
// counter 'n' of a given System PMU 's'.

SPMEVFILTR_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMEVTYPER\_EL0

```
// SPMEVTYPER_EL0[] - non-assignment form
// =====
// Read the current mapping of event with event counter SPMEVCNTR_EL0
// for counter 'n' of a given System PMU 's'.

bits(64) SPMEVTYPER_EL0[integer s, integer n];

// SPMEVTYPER_EL0[] - assignment form
// =====
// Configure which event increments the event counter SPMEVCNTR_EL0, for
// counter 'n' of a given System PMU 's'.

SPMEVTYPER_EL0[integer s, integer n] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMIIDR\_EL1

```
// SPMIIDR_EL1[] - non-assignment form
// =====
// Read the discovery information for System PMU 's'.

bits(64) SPMIIDR_EL1[integer s];
```

### Library pseudocode for aarch64/functions/registers/SPMINTENCLR\_EL1

```
// SPMINTENCLR_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENCLR_EL1[integer s];

// SPMINTENCLR_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters of System PMU 's'.

SPMINTENCLR_EL1[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMINTENSET\_EL1

```
// SPMINTENSET_EL1[] - non-assignment form
// =====
// Read the masking information for interrupt requests on overflows of
// implemented counters of System PMU 's'.

bits(64) SPMINTENSET_EL1[integer s];

// SPMINTENSET_EL1[] - assignment form
// =====
// Disable the generation of interrupt requests on overflows of
// implemented counters for System PMU 's'.

SPMINTENSET_EL1[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMOVSCCLR\_EL0

```
// SPMOVSCCLR_EL0[] - non-assignment form
// =====
// Read the overflow bit clear status of implemented counters for System PMU 's'.

bits(64) SPMOVSCCLR_EL0[integer s];

// SPMOVSCCLR_EL0[] - assignment form
// =====
// Clear the overflow bit clear status of implemented counters for
// System PMU 's'.

SPMOVSCCLR_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMOVSSSET\_EL0

```
// SPMOVSSSET_EL0[] - non-assignment form
// =====
// Read state of the overflow bit for the implemented event counters
// of System PMU 's'.

bits(64) SPMOVSSSET_EL0[integer s];

// SPMOVSSSET_EL0[] - assignment form
// =====
// Sets the state of the overflow bit for the implemented event counters
// of System PMU 's'.

SPMOVSSSET_EL0[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMROOTCR\_EL3

```
// SPMROOTCR_EL3[] - non-assignment form
// =====
// Read the observability of Root and Realm events by System Performance
// Monitor for System PMU 's'.

bits(64) SPMROOTCR_EL3[integer s];

// SPMROOTCR_EL3[] - assignment form
// =====
// Configure the observability of Root and Realm events by System
// Performance Monitor for System PMU 's'.

SPMROOTCR_EL3[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMSCR\_EL1

```
// SPMSCR_EL1[] - non-assignment form
// =====
// Read the observability of Secure events by System Performance Monitor
// for System PMU 's'.

bits(64) SPMSCR_EL1[integer s];

// SPMSCR_EL1[] - assignment form
// =====
// Configure the observability of secure events by System Performance
// Monitor for System PMU 's'.

SPMSCR_EL1[integer s] = bits(64) value;
```

### Library pseudocode for aarch64/functions/registers/SPMZR\_EL0

```
// SPMZR_EL0[] - non-assignment form
// =====
// Read SPMZR_EL0.

bits(64) SPMZR_EL0[integer s];

// SPMZR_EL0[] - assignment form
// =====
// Set event counters for System PMU 's' to zero.

SPMZR_EL0[integer s] = bits(64) value;
```

## Library pseudocode for aarch64/functions/registers/V

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    constant VecLen vlen = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX\_VL);
    else
        _Z[n]<vlen-1:0> = ZeroExtend(value, vlen);

// V[] - non-assignment form
// =====
// Read from SIMD&FP register with implicit slice of 8, 16
// 32, 64 or 128 bits.

bits(width) V[integer n, ESize width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64, 128};
    return _Z[n]<width-1:0>;
```

## Library pseudocode for aarch64/functions/registers/Vpart

```
// Vpart[] - non-assignment form
// =====
// 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.

bits(width) Vpart[integer n, integer part, ESize width]
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        return V[n, width];
    else
        assert width IN {32, 64};
        constant bits(128) vreg = V[n, 128];
        return vreg<(width * 2)-1:width>;

// Vpart[] - assignment form
// =====
// 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.

Vpart[integer n, integer part, ESize width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert part IN {0, 1};
    if part == 0 then
        assert width < 128;
        V[n, width] = value;
    else
        assert width == 64;
        constant bits(64) vreg = V[n, 64];
        V[n, 128] = value<63:0> : vreg;
```

## Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write a 32-bit or 64-bit value to a general-purpose register.

X[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value, 64);
    return;

// X[] - non-assignment form
// =====
// Read the least-significant 8, 16, 32, or 64 bits from a general-purpose register.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8, 16, 32, 64};
    constant rw = width;
    if n != 31 then
        return _R[n]<rw-1:0>;
    else
        return Zeros(rw);
```

## Library pseudocode for aarch64/functions/shiftreg/DecodeShift

```
// DecodeShift()
// =====
// Decode shift encodings

ShiftType DecodeShift(bits(2) op)
    case op of
        when '00' return ShiftType\_LSL;
        when '01' return ShiftType\_LSR;
        when '10' return ShiftType\_ASR;
        when '11' return ShiftType\_ROR;
```

## Library pseudocode for aarch64/functions/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shifttype, integer amount, integer N)
    bits(N) result = X[reg, N];
    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);
    return result;
```

## Library pseudocode for aarch64/functions/shiftreg/ShiftType

```
// ShiftType
// =====
// AArch64 register shifts.

enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

## Library pseudocode for aarch64/functions/sme/CounterToPredicate

```
// CounterToPredicate()
// =====

bits(width) CounterToPredicate(bits(16) pred, integer width)
    integer count;
    ESize esize;
    integer elements;
    constant VecLen VL = CurrentVL;
    constant PredLen PL = VL DIV 8;
    constant integer maxbit = Log2(PL * 4);
    assert maxbit <= 14;
    bits(PL*4) result;
    constant boolean invert = 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;

    elements = (VL * 4) DIV esize;
    result = Zeros(PL*4);
    constant integer psize = esize DIV 8;
    for e = 0 to elements-1
        bit pbit = if e < count then '1' else '0';
        if invert then
            pbit = NOT(pbit);
        Elem[result, e, psize] = ZeroExtend(pbit, psize);

    return result<width-1:0>;
```

## Library pseudocode for aarch64/functions/sme/EncodePredCount

```
// EncodePredCount()
// =====

bits(width) EncodePredCount(ESize esize, integer elements,
                             integer count_in, boolean invert_in, integer width)
    integer count = count_in;
    boolean invert = invert_in;
    constant PredLen PL = CurrentVL DIV 8;
    assert width == PL;
    assert esize IN {8, 16, 32, 64};
    assert count >= 0 && count <= elements;
    bits(16) pred;

    if count == 0 then
        return Zeros(width);

    if invert then
        count = elements - count;
    elsif count == elements then
        count = 0;
        invert = TRUE;

    constant bit inv = (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';

    return ZeroExtend(pred, width);
```

## Library pseudocode for aarch64/functions/sme/Lookup

```
// Lookup Table
// =====

bits(512) _ZT0;
```

## Library pseudocode for aarch64/functions/sme/PredCountTest

```
// PredCountTest()
// =====

bits(4) PredCountTest(integer elements, integer count, boolean invert)
    bit n, z, c, v;
    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
    v = '0';

    return n:z:c:v;
```

## Library pseudocode for aarch64/functions/sme/System

```
// System Registers
// =====

array bits(MAX\_VL) _ZA[0..255];
```

## Library pseudocode for aarch64/functions/sme/ZAhslice

```
// ZAhslice[] - non-assignment form
// =====

bits(width) ZAhslice[integer tile, ESize esize, integer slice, integer width]
    assert esize IN {8, 16, 32, 64, 128};
    constant integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    constant integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    return ZAvector[tile + slice * tiles, width];

// ZAhslice[] - assignment form
// =====

ZAhslice[integer tile, ESize esize, integer slice, integer width] = bits(width) value
    assert esize IN {8, 16, 32, 64, 128};
    constant integer tiles = esize DIV 8;
    assert tile >= 0 && tile < tiles;
    constant integer slices = CurrentSVL DIV esize;
    assert slice >= 0 && slice < slices;

    ZAvector[tile + slice * tiles, width] = value;
```

## Library pseudocode for aarch64/functions/sme/ZAslice

```
// ZAslice[] - non-assignment form
// =====

bits(width) ZAslice[integer tile, ESize esize, boolean vertical, integer slice, integer width]
    bits(width) result;

    if vertical then
        result = ZAvslice[tile, esize, slice, width];
    else
        result = ZAhslice[tile, esize, slice, width];

    return result;

// ZAslice[] - assignment form
// =====

ZAslice[integer tile, ESize esize, boolean vertical,
    integer slice, integer width] = bits(width) value
    if vertical then
        ZAvslice[tile, esize, slice, width] = value;
    else
        ZAhslice[tile, esize, slice, width] = value;
```



## Library pseudocode for aarch64/functions/sme/ZAtile

```
// ZAtile[] - non-assignment form
// =====

bits(width) ZAtile[integer tile, ESize esize, integer width]
    constant VecLen SVL = CurrentSVL;
    constant integer slices = SVL DIV esize;
    assert width == SVL * slices;
    bits(width) result;

    for slice = 0 to slices-1
        Elem[result, slice, SVL] = ZAhslice[tile, esize, slice, SVL];

    return result;

// ZAtile[] - assignment form
// =====

ZAtile[integer tile, ESize esize, integer width] = bits(width) value
    constant VecLen SVL = CurrentSVL;
    constant integer slices = SVL DIV esize;
    assert width == SVL * slices;

    for slice = 0 to slices-1
        ZAhslice[tile, esize, slice, SVL] = Elem[value, slice, SVL];
```

## Library pseudocode for aarch64/functions/sme/ZAvector

```
// ZAvector[] - non-assignment form
// =====

bits(width) ZAvector[integer index, integer width]
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    return \_ZA[index]<width-1:0>;

// ZAvector[] - assignment form
// =====

ZAvector[integer index, integer width] = bits(width) value
    assert width == CurrentSVL;
    assert index >= 0 && index < (width DIV 8);

    if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
        \_ZA[index] = ZeroExtend(value, MAX\_VL);
    else
        \_ZA[index]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sme/ZAvslice

```
// ZAvslice[] - non-assignment form
// =====

bits(width) ZAvslice[integer tile, ESize esize, integer slice, integer width]
    constant integer slices = CurrentSVL DIV esize;
    bits(width) result;

    for s = 0 to slices-1
        constant bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[result, s, esize] = Elem[hslice, slice, esize];

    return result;

// ZAvslice[] - assignment form
// =====

ZAvslice[integer tile, ESize esize, integer slice, integer width] = bits(width) value
    constant integer slices = CurrentSVL DIV esize;

    for s = 0 to slices-1
        bits(width) hslice = ZAhslice[tile, esize, s, width];
        Elem[hslice, slice, esize] = Elem[value, s, esize];
        ZAhslice[tile, esize, s, width] = hslice;
```

## Library pseudocode for aarch64/functions/sme/ZT0

```
// ZT0[] - non-assignment form
// =====

bits(width) ZT0[integer width]
    assert width == 512;
    return _ZT0<width-1:0>;

// ZT0[] - assignment form
// =====

ZT0[integer width] = bits(width) value
    assert width == 512;
    _ZT0<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/AArch32.IsFPEnabled

```
// AArch32.IsFPEnabled()
// =====
// 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.

boolean AArch32.IsFPEnabled(bits(2) el)
    if el == EL0 && !ELUsingAArch32(EL1) then
        return AArch64.IsFPEnabled(el);

    if HaveEL(EL3) && ELUsingAArch32(EL3) && CurrentSecurityState() == SS\_NonSecure then
        // Check if access disabled in NSACR
        if NSACR.cp10 == '0' then return FALSE;

    if el IN {EL0, EL1} then
        // Check if access disabled in CPACR
        boolean disabled;
        case CPACR.cp10 of
            when '00' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '10' disabled = ConstrainUnpredictableBool(Unpredictable\_RESCPACR);
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if !ELUsingAArch32(EL2) then
            return AArch64.IsFPEnabled(EL2);
        if HCPTR.TCP10 == '1' then return FALSE;

    if HaveEL(EL3) && !ELUsingAArch32(EL3) then
        // Check if access disabled in CPTR_EL3
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/AArch64.IsFPEnabled

```
// AArch64.IsFPEnabled()
// =====
// 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.

boolean AArch64.IsFPEnabled(bits(2) el)
    // Check if access disabled in CPACR_EL1
    if el IN {EL0, EL1} && !IsInHost() then
        // Check SIMD&FP at EL0/EL1
        boolean disabled;
        case CPACR_EL1.FPEN of
            when 'x0' disabled = TRUE;
            when '01' disabled = el == EL0;
            when '11' disabled = FALSE;
        if disabled then return FALSE;

    // Check if access disabled in CPTR_EL2
    if el IN {EL0, EL1, EL2} && EL2Enabled() then
        if ELIsInHost(EL2) then
            boolean disabled;
            case CPTR_EL2.FPEN of
                when 'x0' disabled = TRUE;
                when '01' disabled = el == EL0 && HCR_EL2.TGE == '1';
                when '11' disabled = FALSE;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TFP == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.TFP == '1' then return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/ActivePredicateElement

```
// ActivePredicateElement()
// =====
// Returns TRUE if the predicate bit is 1 and FALSE otherwise

boolean ActivePredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    constant integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n> == '1';
```

## Library pseudocode for aarch64/functions/sve/AllElementsActive

```
// AllElementsActive()
// =====
// Return TRUE if all the elements are active in the mask. Otherwise,
// return FALSE.

boolean AllElementsActive(bits(N) mask, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    integer active = 0;
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then active = active + 1;
    return active == elements;
```

## Library pseudocode for aarch64/functions/sve/AnyActiveElement

```
// AnyActiveElement()
// =====
// Return TRUE if there is at least one active element in mask. Otherwise,
// return FALSE.

boolean AnyActiveElement(bits(N) mask, integer esize)
    return LastActiveElement(mask, esize) >= 0;
```

## 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.

bits(N) BitDeposit (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer db = 0;
    for rb = 0 to N-1
        if mask<rb> == '1' then
            res<rb> = data<db>;
            db = db + 1;
    return res;
```

## 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.

bits(N) BitExtract (bits(N) data, bits(N) mask)
    bits(N) res = Zeros(N);
    integer rb = 0;
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

## 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.

bits(N) BitGroup (bits(N) data, bits(N) mask)
    bits(N) res;
    integer rb = 0;

    // compress masked bits to right
    for db = 0 to N-1
        if mask<db> == '1' then
            res<rb> = data<db>;
            rb = rb + 1;
    // compress unmasked bits to left
    for db = 0 to N-1
        if mask<db> == '0' then
            res<rb> = data<db>;
            rb = rb + 1;
    return res;
```

## Library pseudocode for aarch64/functions/sve/CheckNonStreamingSVEEnabled

```
// CheckNonStreamingSVEEnabled()
// =====
// Checks for traps on SVE instructions that are not legal in streaming mode.

CheckNonStreamingSVEEnabled()
    CheckSVEEnabled();

    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' && !IsFullA64Enabled() then
        SMEAccessTrap(SMEExceptionType\_Streaming, PSTATE.EL);
```

## Library pseudocode for aarch64/functions/sve/CheckOriginalSVEEnabled

```
// CheckOriginalSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that access SVE System
// registers.

CheckOriginalSVEEnabled()
    assert IsFeatureImplemented(FEAT_SVE);
    boolean disabled;

    if (HaveEL(EL3) && (CPTR_EL3.EZ == '0' || CPTREL3.TFP == '1') && EL3SDDUndefPriority()) then
        UNDEFINED;

    // 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;
        if disabled then SVEAccessTrap(EL1);

        // 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;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    // 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;
            if disabled then SVEAccessTrap(EL2);

            // 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;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TZ == '1' then SVEAccessTrap(EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SVEAccessTrap(EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then UNDEFINED;
            AArch64.AdvSIMDFPAccessTrap(EL3);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEAccess

```
// CheckSMEAccess()
// =====
// Check that access to SME System registers is enabled.

CheckSMEAccess()
    boolean disabled;

    if HaveEL(EL3) && CPTR_EL3.ESM == '0' && EL3SDDUndefPriority() then
        UNDEFINED;

    // 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;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

    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;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEAndZAAEnabled

```
// CheckSMEAndZAAEnabled()
// =====

CheckSMEAndZAAEnabled()
    CheckSMEEnabled();

    if PSTATE.ZA == '0' then
        SMEAccessTrap(SMEExceptionType_InactiveZA, PSTATE.EL);
```



## Library pseudocode for aarch64/functions/sve/CheckSMEEnabled

```
// CheckSMEEnabled()
// =====

CheckSMEEnabled()
    boolean disabled;

    if HaveEL(EL3) && CPTR_EL3.<ESM,TFP> != '10' && EL3SDDUndefPriority() then
        UNDEFINED;

    // 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;
        if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL1);

        // 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;
        if disabled then AArch64.AdvSIMDFPAccessTrap(EL1);

    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;
            if disabled then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);

            // 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;
            if disabled then AArch64.AdvSIMDFPAccessTrap(EL2);
        else
            if CPTR_EL2.TSM == '1' then SMEAccessTrap(SMEExceptionType_AccessTrap, EL2);
            if CPTR_EL2.TFP == '1' then AArch64.AdvSIMDFPAccessTrap(EL2);

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then
            if EL3SDDUndef() then UNDEFINED;
            SMEAccessTrap(SMEExceptionType_AccessTrap, EL3);

        if CPTR_EL3.TFP == '1' then
            if EL3SDDUndef() then UNDEFINED;
            AArch64.AdvSIMDFPAccessTrap(EL3);
```

## Library pseudocode for aarch64/functions/sve/CheckSMEZT0Enabled

```
// CheckSMEZT0Enabled()
// =====
// Checks for ZT0 enabled.

CheckSMEZT0Enabled()
    if HaveEL\(EL3\) && SMCR_EL3.EZT0 == '0' && EL3SDDUndefPriority\(\) then
        UNDEFINED;

    // Check if ZA and ZT0 are inactive in PSTATE
    if PSTATE.ZA == '0' then
        SMEAccessTrap\(SMEExceptionType\_InactiveZA, PSTATE.EL\);

    // 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\(SMEExceptionType\_InaccessibleZT0, EL1\);

    // 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\(SMEExceptionType\_InaccessibleZT0, EL2\);

    // 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;
            SMEAccessTrap\(SMEExceptionType\_InaccessibleZT0, EL3\);
```

## Library pseudocode for aarch64/functions/sve/CheckSVEEnabled

```
// CheckSVEEnabled()
// =====
// Checks for traps on SVE instructions and instructions that
// access SVE System registers.

CheckSVEEnabled()
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        CheckSMEEnabled\(\);
    elsif IsFeatureImplemented(FEAT_SME) && !IsFeatureImplemented(FEAT_SVE) then
        CheckStreamingSVEEnabled\(\);
    else
        CheckOriginalSVEEnabled\(\);
```

## Library pseudocode for aarch64/functions/sve/CheckStreamingSVEAndZAEEnabled

```
// CheckStreamingSVEAndZAEEnabled()
// =====

CheckStreamingSVEAndZAEEnabled()
    CheckStreamingSVEEnabled\(\);

    if PSTATE.ZA == '0' then
        SMEAccessTrap\(SMEExceptionType\_InactiveZA, PSTATE.EL\);
```

## Library pseudocode for aarch64/functions/sve/CheckStreamingSVEEnabled

```
// CheckStreamingSVEEnabled()
// =====

CheckStreamingSVEEnabled()
    CheckSMEEnabled\(\);

    if PSTATE.SM == '0' then
        SMEAccessTrap\(SMEExceptionType\_NotStreaming, PSTATE.EL\);
```

## Library pseudocode for aarch64/functions/sve/CmpOp

```
// CmpOp
// =====

enumeration CmpOp { Cmp_EQ, Cmp_NE, Cmp_GE, Cmp_GT, Cmp_LT, Cmp_LE, Cmp_UN };
```

## Library pseudocode for aarch64/functions/sve/CurrentNSVL

```
// CurrentNSVL - non-assignment form
// =====
// Non-Streaming VL

VecLen CurrentNSVL
integer vl;

if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
    vl = UInt(ZCR_EL1.LEN);

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));

if PSTATE.EL == EL3 then
    vl = UInt(ZCR_EL3.LEN);
elsif HaveEL(EL3) then
    vl = Min(vl, UInt(ZCR_EL3.LEN));

return ImplementedSVEVectorLength((vl + 1) * 128);
```

## Library pseudocode for aarch64/functions/sve/CurrentSVL

```
// CurrentSVL - non-assignment form
// =====
// Streaming SVL

VecLen CurrentSVL
integer vl;

if PSTATE.EL == EL1 || (PSTATE.EL == EL0 && !IsInHost\(\)) then
    vl = UInt(SMCR_EL1.LEN);

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));

if PSTATE.EL == EL3 then
    vl = UInt(SMCR_EL3.LEN);
elsif HaveEL(EL3) then
    vl = Min(vl, UInt(SMCR_EL3.LEN));

return ImplementedSMEVectorLength((vl + 1) * 128);
```

## Library pseudocode for aarch64/functions/sve/CurrentVL

```
// CurrentVL - non-assignment form
// =====

VecLen CurrentVL
return if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then CurrentSVL else CurrentNSVL;
```

## Library pseudocode for aarch64/functions/sve/DecodePredCount

```
// DecodePredCount()
// =====

integer DecodePredCount(bits(5) bitpattern, integer esize)
    constant integer elements = CurrentVL DIV esize;
    integer numElem;
    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;
    return numElem;
```

## Library pseudocode for aarch64/functions/sve/ElemFFR

```
// ElemFFR[] - non-assignment form
// =====

bit ElemFFR[integer e, ESize esize]
    return PredicateElement(_FFR, e, esize);

// ElemFFR[] - assignment form
// =====

ElemFFR[integer e, ESize esize] = bit value
    constant integer psize = esize DIV 8;
    Elem[_FFR, e, psize] = ZeroExtend(value, psize);
    return;
```

## Library pseudocode for aarch64/functions/sve/FFR

```
// FFR[] - non-assignment form
// =====

bits(width) FFR[integer width]
    assert width == CurrentVL DIV 8;
    return _FFR<width-1:0>;

// FFR[] - assignment form
// =====

FFR[integer width] = bits(width) value
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _FFR = ZeroExtend(value, MAX\_PL);
    else
        _FFR<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/FPCompareNE

```
// FPCompareNE()
// =====

boolean FPCompareNE(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    boolean result;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    op1_nan = type1 IN {FPType\_SNaN, FPType\_QNaN};
    op2_nan = type2 IN {FPType\_SNaN, FPType\_QNaN};

    if op1_nan || op2_nan then
        result = TRUE;
        if type1 == FPType\_SNaN || type2 == FPType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else // All non-NaN cases can be evaluated on the values produced by FPUnpack()
            result = (value1 != value2);
            FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

## Library pseudocode for aarch64/functions/sve/FPCompareUN

```
// FPCompareUN()
// =====

boolean FPCompareUN(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if type1 == FPType\_SNaN || type2 == FPType\_SNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    result = type1 IN {FPType\_SNaN, FPType\_QNaN} || type2 IN {FPType\_SNaN, FPType\_QNaN};
    if !result then
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPConvertSVE

```
// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCR\_Type fpcr_in, FPRounding rounding, integer M)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, rounding, M);

// FPConvertSVE()
// =====

bits(M) FPConvertSVE(bits(N) op, FPCR\_Type fpcr_in, integer M)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```

## Library pseudocode for aarch64/functions/sve/FPExpA

```
// FPExpA()
// =====

bits(N) FPExpA(bits(N) op)
    assert N IN {16,32,64};
    bits(N) result;
    bits(N) coeff;
    constant integer idx = if N == 16 then UInt(op<4:0>) else UInt(op<5:0>);
    coeff = FPExpCoefficient[idx, N];
    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>;

    return result;
```



```

// FPExpCoefficient()
// =====

bits(N) FPExpCoefficient(integer index, integer N)
    assert N IN {16,32,64};
    integer result;

    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;

        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 = 0x0d1adf;
                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;
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;

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 = 0xD5818DCFBA487;
when 57 result = 0xDA9E603DB3285;
when 58 result = 0xDFC97337B9B5F;
when 59 result = 0xE502EE78B3FF6;
when 60 result = 0xEA4AFA2A490DA;
when 61 result = 0xEFA1BEE615A27;
when 62 result = 0xF50765B6E4540;
when 63 result = 0xFA7C1819E90D8;

```

```
return result<N-1:0>;
```

## Library pseudocode for aarch64/functions/sve/FPLogB

```

// FPLogB()
// =====

bits(N) FPLogB(bits(N) op, FPCR\_Type fpcr)
  assert N IN {16,32,64};
  integer result;
  (fptype,sign,value) = FPUnpack(op, fpcr);

  if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN || fptype == FPTYPE\_Zero then
    FPProcessException(FPExc\_InvalidOp, fpcr);
    result = -(2^(N-1)); // MinInt, 100..00
  elsif fptype == FPTYPE\_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);
  return result<N-1:0>;

```

## Library pseudocode for aarch64/functions/sve/FPMinNormal

```
// FPMinNormal()
// =====

bits(N) FPMinNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Zeros(E-1):'1';
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPOne

```
// FPOne()
// =====

bits(N) FPOne(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPPointFive

```
// FPPointFive()
// =====

bits(N) FPPointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-2):'0';
    frac = Zeros(F);
    return sign : exp : frac;
```

## Library pseudocode for aarch64/functions/sve/FPReducePredicated

```
// FPReducePredicated()
// =====

bits(esize) FPReducePredicated(ReduceOp op, bits(N) input, bits(M) mask,
                                bits(esize) identity, FPCR\_Type fpcr)
    assert(N == M * 8);
    assert IsPow2(N);
    bits(N) operand;
    constant integer elements = N DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return FPReduce(op, operand, esize, fpcr);
```

## Library pseudocode for aarch64/functions/sve/FPTrigMAdd

```
// FPTrigMAdd()
// =====

bits(N) FPTrigMAdd(integer x_in, bits(N) op1, bits(N) op2_in, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    bits(N) coeff;
    bits(N) op2 = op2_in;
    integer x = x_in;
    assert x >= 0;
    assert x < 8;

    if op2<N-1> == '1' then
        x = x + 8;

    coeff    = FPTrigMAddCoefficient[x, N];
    // Safer to use EffectiveFPCR() in case the input fpcr argument
    // is modified as opposed to actual value of FPCR

    op2      = FPAbs(op2, EffectiveFPCR());
    result   = FPMulAdd(coeff, op1, op2, fpcr);
    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigMAddCoefficient

```
// FPTrigMAddCoefficient()
// =====

bits(N) FPTrigMAddCoefficient[integer index, integer N]
  assert N IN {16,32,64};
  integer result;

  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;
    elseif N == 32 then
      case index of
        when 0 result = 0x3f800000;
        when 1 result = 0xbe2aaaab;
        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 = 0x3d2aaaa6;
        when 11 result = 0xbab60705;
        when 12 result = 0x37cd37cc;
        when 13 result = 0x00000000;
        when 14 result = 0x00000000;
        when 15 result = 0x00000000;
      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;

  return result<N-1:0>;
```

## Library pseudocode for aarch64/functions/sve/FPTrigSMul

```
// FPTrigSMul()
// =====

bits(N) FPTrigSMul(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    result = FPMul(op1, op1, fpcr);
    fpexc = FALSE;
    (fptype, sign, value) = FPUnpack(result, fpcr, fpexc);

    if ! fptype IN {FPType\_QNaN, FPType\_SNaN} then
        result<N-1> = op2<0>;

    return result;
```

## Library pseudocode for aarch64/functions/sve/FPTrigSSel

```
// FPTrigSSel()
// =====

bits(N) FPTrigSSel(bits(N) op1, bits(N) op2)
    assert N IN {16,32,64};
    bits(N) result;

    if op2<0> == '1' then
        result = FPOne(op2<1>, N);
    elsif op2<1> == '1' then
        result = FPNeg(op1, EffectiveFPCR());
    else
        result = op1;

    return result;
```

## Library pseudocode for aarch64/functions/sve/FirstActive

```
// FirstActive()
// =====

bit FirstActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

## Library pseudocode for aarch64/functions/sve/Getter

```
// Getter for SVCR
// =====
// Returns PSTATE.<ZA, SM>

SVCR_Type SVCR
    constant SVCR_Type value = Zeros(62) : PSTATE.ZA : PSTATE.SM;
    return value;
```

## Library pseudocode for aarch64/functions/sve/HaveSVE2FP8DOT2

```
// HaveSVE2FP8DOT2()
// =====
// Returns TRUE if SVE2 FP8 dot product to half-precision instructions
// are implemented, FALSE otherwise.

boolean HaveSVE2FP8DOT2()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT2)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT2));
```

### Library pseudocode for aarch64/functions/sve/HaveSVE2FP8DOT4

```
// HaveSVE2FP8DOT4()
// =====
// Returns TRUE if SVE2 FP8 dot product to single-precision instructions
// are implemented, FALSE otherwise.

boolean HaveSVE2FP8DOT4()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8DOT4)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8DOT4));
```

### 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.

boolean HaveSVE2FP8FMA()
    return ((IsFeatureImplemented(FEAT_SVE2) && IsFeatureImplemented(FEAT_FP8FMA)) ||
            IsFeatureImplemented(FEAT_SSVE_FP8FMA));
```

### Library pseudocode for aarch64/functions/sve/ImplementedSMEVectorLength

```
// ImplementedSMEVectorLength()
// =====
// Reduce SVE/SME vector length to a supported value (power of two)

VecLen ImplementedSMEVectorLength(integer nbits_in)
    constant VecLen maxbits = MaxImplementedSVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    // Search for a supported power-of-two VL less than or equal to nbits
    while nbits > 128 && !SupportedPowerTwoSVL(nbits) do
        nbits = nbits - 128;

    // Return the smallest supported power-of-two VL
    while nbits < maxbits && !SupportedPowerTwoSVL(nbits) do
        nbits = nbits * 2;

    return nbits;
```

### Library pseudocode for aarch64/functions/sve/ImplementedSVEVectorLength

```
// ImplementedSVEVectorLength()
// =====
// Reduce SVE vector length to a supported value (power of two)

VecLen ImplementedSVEVectorLength(integer nbits_in)
    constant integer maxbits = MaxImplementedVL();
    assert 128 <= maxbits && maxbits <= 2048 && IsPow2(maxbits);
    integer nbits = Min(nbits_in, maxbits);
    assert 128 <= nbits && nbits <= 2048 && Align(nbits, 128) == nbits;

    while !IsPow2(nbits) do
        nbits = nbits - 128;
    return nbits;
```

### Library pseudocode for aarch64/functions/sve/InStreamingMode

```
// InStreamingMode()
// =====

boolean InStreamingMode()
    return IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1';
```

## Library pseudocode for aarch64/functions/sve/IntReducePredicated

```
// IntReducePredicated()
// =====

bits(esize) IntReducePredicated(ReduceOp op, bits(N) input, bits(M) mask, bits(esize) identity)
    assert(N == M * 8);
    assert IsPow2(N);
    bits(N) operand;
    constant integer elements = N DIV esize;

    for e = 0 to elements-1
        if e * esize < N && ActivePredicateElement(mask, e, esize) then
            Elem[operand, e, esize] = Elem[input, e, esize];
        else
            Elem[operand, e, esize] = identity;

    return IntReduce(op, operand, esize);
```

## 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.

boolean IsFPEnabled(bits(2) el)
    if ELUsingAArch32(el) then
        return AArch32.IsFPEnabled(el);
    else
        return AArch64.IsFPEnabled(el);
```

## Library pseudocode for aarch64/functions/sve/IsFullA64Enabled

```
// IsFullA64Enabled()
// =====
// Returns TRUE if full A64 is enabled in Streaming mode and FALSE otherwise.

boolean IsFullA64Enabled()
    if !IsFeatureImplemented(FEAT_SME_FA64) then return FALSE;

    // 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;

    // 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;

    // Check if full A64 disabled in SMCR_EL3
    if HaveEL(EL3) then
        if SMCR_EL3.FA64 == '0' then return FALSE;

    return TRUE;
```



## 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.

boolean IsOriginalSVEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // 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;
        if disabled then return FALSE;

    // 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;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TZ == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.EZ == '0' then return FALSE;

    return TRUE;
```

## 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.

boolean IsSMEEnabled(bits(2) el)
    boolean disabled;
    if ELUsingAArch32(el) then
        return FALSE;

    // 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;
        if disabled then return FALSE;

    // 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;
            if disabled then return FALSE;
        else
            if CPTR_EL2.TSM == '1' then return FALSE;

    // Check if access disabled in CPTR_EL3
    if HaveEL(EL3) then
        if CPTR_EL3.ESM == '0' then return FALSE;

    return TRUE;
```

## 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.

boolean IsSVEEnabled(bits(2) el)
    if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        return IsSMEEnabled(el);
    elseif IsFeatureImplemented(FEAT_SVE) then
        return IsOriginalSVEEnabled(el);
    else
        return FALSE;
```

## Library pseudocode for aarch64/functions/sve/LastActive

```
// LastActive()
// =====

bit LastActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then
            return PredicateElement(x, e, esize);
    return '0';
```

### Library pseudocode for aarch64/functions/sve/LastActiveElement

```
// LastActiveElement()
// =====

integer LastActiveElement(bits(N) mask, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = elements-1 downto 0
        if ActivePredicateElement(mask, e, esize) then return e;
    return -1;
```

### Library pseudocode for aarch64/functions/sve/MaxImplementedAnyVL

```
// MaxImplementedAnyVL()
// =====

integer MaxImplementedAnyVL()
    if IsFeatureImplemented(FEAT_SME) && IsFeatureImplemented(FEAT_SVE) then
        return Max(MaxImplementedVL(), MaxImplementedSVL());
    if IsFeatureImplemented(FEAT_SME) then
        return MaxImplementedSVL();
    return MaxImplementedVL();
```

### Library pseudocode for aarch64/functions/sve/MaxImplementedSVL

```
// MaxImplementedSVL()
// =====

VecLen MaxImplementedSVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented SVL";
```

### Library pseudocode for aarch64/functions/sve/MaxImplementedVL

```
// MaxImplementedVL()
// =====

integer MaxImplementedVL()
    return integer IMPLEMENTATION_DEFINED "Max implemented VL";
```

## Library pseudocode for aarch64/functions/sve/MaybeZeroSVEUppers

```
// MaybeZeroSVEUppers()
// =====

MaybeZeroSVEUppers(bits(2) target_el)
    boolean lower_enabled;

    if UInt(target_el) <= UInt(PSTATE.EL) || !IsSVEEnabled(target_el) then
        return;

    if target_el == EL3 then
        if EL2Enabled() then
            lower_enabled = IsFPEEnabled(EL2);
        else
            lower_enabled = IsFPEEnabled(EL1);
    elseif target_el == EL2 then
        assert EL2Enabled() && !ELUsingAArch32(EL2);
        if HCR_EL2.TGE == '0' then
            lower_enabled = IsFPEEnabled(EL1);
        else
            lower_enabled = IsFPEEnabled(EL0);
    else
        assert target_el == EL1 && !ELUsingAArch32(EL1);
        lower_enabled = IsFPEEnabled(EL0);

    if lower_enabled then
        constant integer VL = if IsSVEEnabled(PSTATE.EL) then CurrentVL else 128;
        constant integer PL = VL DIV 8;
        for n = 0 to 31
            if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
                _Z[n] = ZeroExtend(_Z[n]<VL-1:0>, MAX_VL);
        for n = 0 to 15
            if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
                _P[n] = ZeroExtend(_P[n]<PL-1:0>, MAX_PL);
        if ConstrainUnpredictableBool(Unpredictable SVEZEROUPPER) then
            _FFR = ZeroExtend(_FFR<PL-1:0>, MAX_PL);
        if IsFeatureImplemented(FEAT_SME) && PSTATE.ZA == '1' then
            constant integer SVL = CurrentSVL;
            constant integer accessiblevecs = SVL DIV 8;
            constant integer allvecs = MaxImplementedSVL() DIV 8;

            for n = 0 to accessiblevecs - 1
                if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
                    _ZA[n] = ZeroExtend(_ZA[n]<SVL-1:0>, MAX_VL);
            for n = accessiblevecs to allvecs - 1
                if ConstrainUnpredictableBool(Unpredictable SMEZEROUPPER) then
                    _ZA[n] = Zeros(MAX_VL);
```

## Library pseudocode for aarch64/functions/sve/MemNF

```
// MemNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemNF[bits(64) address, integer size, AccessDescriptor accdesc]
    assert size IN {1, 2, 4, 8, 16};
    bits(8*size) value;
    boolean bad;

    boolean aligned = IsAligned(address, size);

    if !aligned && AlignmentEnforced() then
        return (bits(8*size) UNKNOWN, TRUE);

    constant boolean atomic = aligned || size == 1;

    if !atomic then
        (value<7:0>, bad) = MemSingleNF[address, 1, accdesc, aligned];

        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

        // 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
            c = ConstrainUnpredictable(Unpredictable\_DEVPAGE2);
            assert c IN {Constraint\_FAULT, Constraint\_NONE};
            if c == Constraint\_NONE then aligned = TRUE;

        for i = 1 to size-1
            (Elem[value, i, 8], bad) = MemSingleNF[address+i, 1, accdesc, aligned];

            if bad then
                return (bits(8*size) UNKNOWN, TRUE);
    else
        (value, bad) = MemSingleNF[address, size, accdesc, aligned];
        if bad then
            return (bits(8*size) UNKNOWN, TRUE);

    if BigEndian(accdesc.acctype) then
        value = BigEndianReverse(value);

    return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/MemSingleNF

```
// MemSingleNF[] - non-assignment form
// =====

(bits(8*size), boolean) MemSingleNF(bits(64) address, integer size, AccessDescriptor accdesc_in,
                                     boolean aligned]
    assert accdesc_in.acctype == AccessType\_SVE;
    assert accdesc_in.nonfault || (accdesc_in.firstfault && !accdesc_in.first);

    bits(8*size) value;
    AddressDescriptor memaddrdesc;
    PhysMemRetStatus memstatus;
    AccessDescriptor accdesc = accdesc_in;
    FaultRecord fault = NoFault(accdesc, address);

    // Implementation may suppress NF load for any reason
    if ConstrainUnpredictableBool(Unpredictable\_NONFAULT) then
        return (bits(8*size) UNKNOWN, TRUE);

    // If the instruction encoding permits tag checking, confer with system register configuration
    // which may override this.
    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        accdesc.tagchecked = AArch64.AccessIsTagChecked(address, accdesc);

    // MMU or MPU
    memaddrdesc = AArch64.TranslateAddress(address, accdesc, aligned, size);

    // Non-fault load from Device memory must not be performed externally
    if memaddrdesc.memattrs.memtype == MemType\_Device then
        return (bits(8*size) UNKNOWN, TRUE);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return (bits(8*size) UNKNOWN, TRUE);

    if IsFeatureImplemented(FEAT_MTE2) && accdesc.tagchecked then
        constant bits(4) ltag = AArch64.LogicalAddressTag(address);
        if (!AArch64.CheckTag(memaddrdesc, accdesc, ltag) &&
            AArch64.EffectiveTCF(accdesc.el, accdesc.read) != TCFType\_Ignore) then
            return (bits(8*size) UNKNOWN, TRUE);

    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        constant boolean iswrite = FALSE;
        if IsExternalAbortTakenSynchronously(memstatus, iswrite, memaddrdesc, size, accdesc) then
            return (bits(8*size) UNKNOWN, TRUE);
        fault.merrorstate = memstatus.merrorstate;
        fault.extflag = memstatus.extflag;
        fault.statuscode = memstatus.statuscode;
        PendSErrorInterrupt(fault);

    return (value, FALSE);
```

## Library pseudocode for aarch64/functions/sve/NoneActive

```
// NoneActive()
// =====

bit NoneActive(bits(N) mask, bits(N) x, integer esize)
    constant integer elements = N DIV (esize DIV 8);
    for e = 0 to elements-1
        if ActivePredicateElement(mask, e, esize) && ActivePredicateElement(x, e, esize) then
            return '0';
    return '1';
```

## Library pseudocode for aarch64/functions/sve/P

```
// P[] - non-assignment form
// =====

bits(width) P[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    return _P[n]<width-1:0>;

// P[] - assignment form
// =====

P[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL DIV 8;
    if ConstrainUnpredictableBool(Unpredictable SVEZERoupper) then
        _P[n] = ZeroExtend(value, MAX\_PL);
    else
        _P[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/sve/PredLen

```
// PredLen
// =====

type PredLen = integer;
```

## Library pseudocode for aarch64/functions/sve/PredTest

```
// PredTest()
// =====

bits(4) PredTest(bits(N) mask, bits(N) result, integer esize)
    constant bit n = FirstActive(mask, result, esize);
    constant bit z = NoneActive(mask, result, esize);
    constant bit c = NOT LastActive(mask, result, esize);
    constant bit v = '0';
    return n:z:c:v;
```

## Library pseudocode for aarch64/functions/sve/PredicateElement

```
// PredicateElement()
// =====
// Returns the predicate bit

bit PredicateElement(bits(N) pred, integer e, integer esize)
    assert esize IN {8, 16, 32, 64, 128};
    constant integer n = e * (esize DIV 8);
    assert n >= 0 && n < N;
    return pred<n>;
```

## Library pseudocode for aarch64/functions/sve/ResetSMEState

```
// ResetSMEState()
// =====

ResetSMEState(bit newenable)
    constant integer vectors = MAX\_VL DIV 8;
    if newenable == '1' then
        for n = 0 to vectors - 1
            _ZA[n] = Zeros(MAX\_VL);
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = Zeros(ZT0\_LEN);
    else
        for n = 0 to vectors - 1
            _ZA[n] = bits(MAX\_VL) UNKNOWN;
        if IsFeatureImplemented(FEAT_SME2) then
            _ZT0 = bits(ZT0\_LEN) UNKNOWN;
```

## Library pseudocode for aarch64/functions/sve/ResetSVERegisters

```
// ResetSVERegisters()
// =====

ResetSVERegisters()
    for n = 0 to 31
        _Z[n] = bits(MAX\_VL) UNKNOWN;
    for n = 0 to 15
        _P[n] = bits(MAX\_PL) UNKNOWN;
    _FFR = bits(MAX\_PL) UNKNOWN;
```

## Library pseudocode for aarch64/functions/sve/ResetSVEState

```
// ResetSVEState()
// =====

ResetSVEState()
    for n = 0 to 31
        _Z[n] = Zeros(MAX\_VL);
    for n = 0 to 15
        _P[n] = Zeros(MAX\_PL);
    _FFR = Zeros(MAX\_PL);
    FPSR = ZeroExtend(0x0800009f<31:0>, 64);
    FPMR = Zeros(64);
```



## Library pseudocode for aarch64/functions/sve/SMEAccessTrap

```
// SMEAccessTrap()
// =====
// Trapped access to SME registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SMEAccessTrap(SMEExceptionType etype, bits(2) target_el_in)
    bits(2) target_el = target_el_in;
    assert UInt(target_el) >= UInt(PSTATE.EL);
    if target_el == EL0 then
        target_el = EL1;
    boolean route_to_el2;
    route_to_el2 = PSTATE.EL == EL0 && target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SMEAccessTrap);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 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';

    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);
```

## Library pseudocode for aarch64/functions/sve/SMEExceptionType

```
// SMEExceptionType
// =====
enumeration SMEExceptionType {
    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/SVEAccessTrap

```
// SVEAccessTrap()
// =====
// Trapped access to SVE registers due to CPACR_EL1, CPTR_EL2, or CPTR_EL3.

SVEAccessTrap(bits(2) target_el)
    assert UInt(target_el) >= UInt(PSTATE.EL) && target_el != EL0 && HaveEL(target_el);
    route_to_el2 = target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1';

    except = ExceptionSyndrome(Exception_SVEAccessTrap);
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 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);
```

## 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.

boolean SVEMoveMaskPreferred(bits(13) imm13)
    bits(64) imm;
    (imm, -) = DecodeBitMasks(imm13<12>, imm13<5:0>, imm13<11:6>, TRUE, 64);

    // Check for 8 bit immediates
    if !IsZero(imm<7:0>) then
        // Check for 'ffffffffffffffxy' or '00000000000000xy'
        if IsZero(imm<63:7>) || IsOnes(imm<63:7>) then
            return FALSE;

        // Check for 'ffffffxyffffffxy' or '000000xy000000xy'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // 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;

        // 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;

    // Check for 16 bit immediates
    else
        // Check for 'ffffffffffffxy00' or '000000000000xy00'
        if IsZero(imm<63:15>) || IsOnes(imm<63:15>) then
            return FALSE;

        // Check for 'ffffxy00ffffxy00' or '0000xy000000xy00'
        if imm<63:32> == imm<31:0> && (IsZero(imm<31:7>) || IsOnes(imm<31:7>)) then
            return FALSE;

        // Check for 'xy00xy00xy00xy00'
        if imm<63:32> == imm<31:0> && imm<31:16> == imm<15:0> then
            return FALSE;

    return TRUE;
```

## Library pseudocode for aarch64/functions/sve/SetPSTATE\_SM

```
// SetPSTATE_SM()
// =====

SetPSTATE_SM(bit value)
    if PSTATE.SM != value then
        ResetSVEState();
        PSTATE.SM = value;
```

## Library pseudocode for aarch64/functions/sve/SetPSTATE\_ZA

```
// SetPSTATE_ZA()
// =====

SetPSTATE_ZA(bit value)
    if PSTATE.ZA != value then
        ResetSMEState(value);
        PSTATE.ZA = value;
```

### Library pseudocode for aarch64/functions/sve/Setter

```
// Setter for SVCR
// =====
// Sets PSTATE.<ZA, SM>

SVCR = SVCR_Type value
  SetPSTATE\_SM(value<0>);
  SetPSTATE\_ZA(value<1>);
return;
```

### Library pseudocode for aarch64/functions/sve/ShiftSat

```
// ShiftSat()
// =====

integer ShiftSat(integer shift, integer esize)
  if shift > esize+1 then return esize+1;
  elsif shift < -(esize+1) then return -(esize+1);
  return shift;
```

### 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

boolean SupportedPowerTwoSVL(integer nbits);
```

### Library pseudocode for aarch64/functions/sve/System

```
// System Registers
// =====

constant integer MAX_VL = 2048;
constant integer MAX_PL = 256;
constant integer ZT0_LEN = 512;
bits(MAX\_PL) _FFR;

array bits(MAX\_VL) _Z[0..31];

array bits(MAX\_PL) _P[0..15];
```

### Library pseudocode for aarch64/functions/sve/VecLen

```
// VecLen
// =====

type VecLen = integer;
```

## Library pseudocode for aarch64/functions/sve/Z

```
// Z[] - non-assignment form
// =====

bits(width) Z[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    return _Z[n]<width-1:0>;

// Z[] - assignment form
// =====

Z[integer n, integer width] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width == CurrentVL;
    if ConstrainUnpredictableBool(Unpredictable\_SVEZEROUPPER) then
        _Z[n] = ZeroExtend(value, MAX\_VL);
    else
        _Z[n]<width-1:0> = value;
```

## Library pseudocode for aarch64/functions/syshintop/SystemHintOp

```
// SystemHintOp
// =====
// System Hint instruction types.

enumeration SystemHintOp {
    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
};
```



```

// SysOp()
// =====

SystemOp SysOp(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  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;    // S12E1R
    when '100 0111 1000 101' return Sys AT;    // S12E1W
    when '100 0111 1000 110' return Sys AT;    // S12E0R
    when '100 0111 1000 111' return Sys AT;    // S12E0W
    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;   // VMALLE1OS
    when '000 1000 0001 001' return Sys TLBI;   // VAE1OS
    when '000 1000 0001 010' return Sys TLBI;   // ASIDE1OS
    when '000 1000 0001 011' return Sys TLBI;   // VAAE1OS
    when '000 1000 0001 101' return Sys TLBI;   // VALE1OS
    when '000 1000 0001 111' return Sys TLBI;   // VAALE1OS
    when '000 1000 0010 001' return Sys TLBI;   // RVAE1IS
    when '000 1000 0010 011' return Sys TLBI;   // RVAAE1IS
    when '000 1000 0010 101' return Sys TLBI;   // RVALE1IS

```

```

when '000 1000 0010 111' return Sys TLBI; // RVAALE1IS
when '000 1000 0011 000' return Sys TLBI; // VMALLE1IS
when '000 1000 0011 001' return Sys TLBI; // VAE1IS
when '000 1000 0011 010' return Sys TLBI; // ASIDE1IS
when '000 1000 0011 011' return Sys TLBI; // VAAE1IS
when '000 1000 0011 101' return Sys TLBI; // VALE1IS
when '000 1000 0011 111' return Sys TLBI; // VAALE1IS
when '000 1000 0101 001' return Sys TLBI; // RVAE1OS
when '000 1000 0101 011' return Sys TLBI; // RVAAE1OS
when '000 1000 0101 101' return Sys TLBI; // RVALE1OS
when '000 1000 0101 111' return Sys TLBI; // RVAALE1OS
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
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; // RVAALE1ISNXS
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; // RVAALE1OSNXS
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; // RVAALE1NXS
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;

```

## Library pseudocode for aarch64/functions/sysop/SystemOp

```

// SystemOp
// =====
// System instruction types.

enumeration SystemOp {Sys_AT, Sys_BRB, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};

```



```

// SysOp128()
// =====

SystemOp128 SysOp128(bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2)
  case op1:CRn:CRm:op2 of
    when '000 1000 0001 001' return Sys_TLBIP; // VAE1OS
    when '000 1000 0001 011' return Sys_TLBIP; // VAAE1OS
    when '000 1000 0001 101' return Sys_TLBIP; // VALE1OS
    when '000 1000 0001 111' return Sys_TLBIP; // VAALE1OS
    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; // VAE1OSNXS
    when '000 1001 0001 011' return Sys_TLBIP; // VAAE1OSNXS
    when '000 1001 0001 101' return Sys_TLBIP; // VALE1OSNXS
    when '000 1001 0001 111' return Sys_TLBIP; // VAALE1OSNXS
    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; // VAE2OS
    when '100 1000 0001 101' return Sys_TLBIP; // VALE2OS
    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; // VAE2OSNXS
    when '100 1001 0001 101' return Sys_TLBIP; // VALE2OSNXS
    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; // VAE3OS
    when '110 1000 0001 101' return Sys_TLBIP; // VALE3OS
    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; // VAE3OSNXS
    when '110 1001 0001 101' return Sys_TLBIP; // VALE3OSNXS
    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; // IPAS2E1OS
    when '100 1000 0100 001' return Sys_TLBIP; // IPAS2E1
    when '100 1000 0100 100' return Sys_TLBIP; // IPAS2LE1OS
    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; // IPAS2E1OSNXS
    when '100 1001 0100 001' return Sys_TLBIP; // IPAS2E1NXS
    when '100 1001 0100 100' return Sys_TLBIP; // IPAS2LE1OSNXS
    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; // RVAE1OS

```

```

when '000 1000 0101 011' return Sys_TLBIP; // RVAAE1OS
when '000 1000 0101 101' return Sys_TLBIP; // RVALE1OS
when '000 1000 0101 111' return Sys_TLBIP; // RVAALE1OS
when '000 1000 0110 001' return Sys_TLBIP; // RVAE1
when '000 1000 0110 011' return Sys_TLBIP; // RVAAE1
when '000 1000 0110 101' return Sys_TLBIP; // RVALE1
when '000 1000 0110 111' return Sys_TLBIP; // RVAALE1
when '000 1001 0010 001' return Sys_TLBIP; // RVAE1ISNXS
when '000 1001 0010 011' return Sys_TLBIP; // RVAAE1ISNXS
when '000 1001 0010 101' return Sys_TLBIP; // RVALE1ISNXS
when '000 1001 0010 111' return Sys_TLBIP; // RVAALE1ISNXS
when '000 1001 0101 001' return Sys_TLBIP; // RVAE1OSNXS
when '000 1001 0101 011' return Sys_TLBIP; // RVAAE1OSNXS
when '000 1001 0101 101' return Sys_TLBIP; // RVALE1OSNXS
when '000 1001 0101 111' return Sys_TLBIP; // RVAALE1OSNXS
when '000 1001 0110 001' return Sys_TLBIP; // RVAE1NXS
when '000 1001 0110 011' return Sys_TLBIP; // RVAAE1NXS
when '000 1001 0110 101' return Sys_TLBIP; // RVALE1NXS
when '000 1001 0110 111' return Sys_TLBIP; // RVAALE1NXS
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;

```

## Library pseudocode for aarch64/functions/sysop\_128/SystemOp128

```

// SystemOp128()
// =====
// System instruction types.

enumeration SystemOp128 {Sys_TLBIP, Sys_SYSP};

```

## Library pseudocode for aarch64/functions/sysregisters/ELR\_EL

```
// ELR_EL[] - non-assignment form
// =====

bits(64) ELR_EL[bits(2) el]
  bits(64) r;
  case el of
    when EL1   r = ELR_EL1;
    when EL2   r = ELR_EL2;
    when EL3   r = ELR_EL3;
    otherwise Unreachable();
  return r;

// ELR_EL[] - assignment form
// =====

ELR_EL[bits(2) el] = bits(64) value
  constant bits(64) r = value;
  case el of
    when EL1   ELR_EL1 = r;
    when EL2   ELR_EL2 = r;
    when EL3   ELR_EL3 = r;
    otherwise Unreachable();
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ELR\_ELx

```
// ELR_ELx[] - non-assignment form
// =====

bits(64) ELR_ELx[]
  assert PSTATE.EL != EL0;
  return ELR\_EL[PSTATE.EL];

// ELR_ELx[] - assignment form
// =====

ELR_ELx[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR\_EL[PSTATE.EL] = value;
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ESR\_EL

```
// ESR_EL[] - non-assignment form
// =====

ESRType ESR_EL[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = ESR_EL1;
    when EL2   r = ESR_EL2;
    when EL3   r = ESR_EL3;
    otherwise Unreachable();
  return r;

// ESR_EL[] - assignment form
// =====

ESR_EL[bits(2) regime] = ESRType value
  constant bits(64) r = value;
  case regime of
    when EL1   ESR_EL1 = r;
    when EL2   ESR_EL2 = r;
    when EL3   ESR_EL3 = r;
    otherwise Unreachable();
  return;
```

## Library pseudocode for aarch64/functions/sysregisters/ESR\_ELx

```
// ESR_ELx[] - non-assignment form
// =====

ESRType ESR_ELx[]
    return ESR_EL[S1TranslationRegime()];

// ESR_ELx[] - assignment form
// =====

ESR_ELx[] = ESRType value
    ESR_EL[S1TranslationRegime()] = value;
```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_EL

```
// FAR_EL[] - non-assignment form
// =====

bits(64) FAR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1    r = FAR_EL1;
        when EL2    r = FAR_EL2;
        when EL3    r = FAR_EL3;
        otherwise Unreachable();
    return r;

// FAR_EL[] - assignment form
// =====

FAR_EL[bits(2) regime] = bits(64) value
    constant bits(64) r = value;
    case regime of
        when EL1    FAR_EL1 = r;
        when EL2    FAR_EL2 = r;
        when EL3    FAR_EL3 = r;
        otherwise Unreachable();
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/FAR\_ELx

```
// FAR_ELx[] - non-assignment form
// =====

bits(64) FAR_ELx[]
    return FAR_EL[S1TranslationRegime()];

// FAR_ELx[] - assignment form
// =====

FAR_ELx[] = bits(64) value
    FAR_EL[S1TranslationRegime()] = value;
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/PFAR\_EL

```
// PFAR_EL[] - non-assignment form
// =====

bits(64) PFAR_EL[bits(2) regime]
    assert (IsFeatureImplemented(FEAT_PFAR) || (regime == EL3 && IsFeatureImplemented(FEAT_RME)));
    bits(64) r;
    case regime of
        when EL1 r = PFAR_EL1;
        when EL2 r = PFAR_EL2;
        when EL3 r = MFAR_EL3;
        otherwise Unreachable();
    return r;

// PFAR_EL[] - assignment form
// =====

PFAR_EL[bits(2) regime] = bits(64) value
    constant bits(64) r = value;
    assert (IsFeatureImplemented(FEAT_PFAR) || (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();
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/PFAR\_ELx

```
// PFAR_ELx[] - non-assignment form
// =====

bits(64) PFAR_ELx[]
    return PFAR\_EL\[S1TranslationRegime\(\)\];

// PFAR_ELx[] - assignment form
// =====

PFAR_ELx[] = bits(64) value
    PFAR\_EL\[S1TranslationRegime\(\)\] = value;
    return;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLR\_EL

```
// SCTLR_EL[] - non-assignment form
// =====

SCTLRType SCTLR_EL[bits(2) regime]
    bits(64) r;
    case regime of
        when EL1 r = SCTLR_EL1;
        when EL2 r = SCTLR_EL2;
        when EL3 r = SCTLR_EL3;
        otherwise Unreachable();
    return r;
```

## Library pseudocode for aarch64/functions/sysregisters/SCTLR\_ELx

```
// SCTLR_ELx[] - non-assignment form
// =====

SCTLRType SCTLR_ELx[]
    return SCTLR_EL[S1TranslationRegime\(\)];
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR\_EL

```
// VBAR_EL[] - non-assignment form
// =====

bits(64) VBAR_EL[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1   r = VBAR_EL1;
    when EL2   r = VBAR_EL2;
    when EL3   r = VBAR_EL3;
    otherwise Unreachable();
  return r;
```

## Library pseudocode for aarch64/functions/sysregisters/VBAR\_ELx

```
// VBAR_ELx[] - non-assignment form
// =====

bits(64) VBAR_ELx[]
  return VBAR\_EL[S1TranslationRegime()];
```

## Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(bits(2) el)
  if !IsFeatureImplemented(FEAT_MTE2) then return FALSE;

  if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
    return FALSE;
  if HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && !ELIsInHost(EL0) then
    return FALSE;

  constant Regime regime = TranslationRegime(el);
  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();
```

## Library pseudocode for aarch64/functions/system/AArch64.CheckDAIFAccess

```
// AArch64.CheckDAIFAccess()
// =====
// Check that an AArch64 MSR/MRS access to the DAIF flags is permitted.

AArch64.CheckDAIFAccess(PSTATEField field)
  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);
```



### Library pseudocode for aarch64/functions/system/AArch64.CheckSystemAccess

```
// AArch64.CheckSystemAccess()
// =====

AArch64.CheckSystemAccess(integer op0, integer op1, integer crn,
                           integer crm, integer op2, integer rt, integer read)
    if (IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 &&
        !CheckTransactionalSystemAccess(op0, op1, crn, crm, op2, read)) then
        FailTransaction(TMFailure_ERR, FALSE);

    return;
```

### 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.

bits(4) AArch64.ChooseNonExcludedTag(bits(4) tag_in, bits(4) offset_in, bits(16) exclude)
    bits(4) tag = tag_in;
    bits(4) offset = offset_in;

    if IsOnes(exclude) then
        return '0000';

    if offset == '0000' then
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    while offset != '0000' do
        offset = offset - '0001';
        tag = tag + '0001';
        while exclude<UInt(tag)> == '1' do
            tag = tag + '0001';

    return tag;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !IsFeatureImplemented(FEAT_BTI) then return FALSE;

    instr = ThisInstr();
    if instr<31:22> == '1101010100' && instr<21:12> == '0000110010' && instr<4:0> == '11111' then
        CRm = instr<11:8>;
        op2 = instr<7:5>;
        return (CRm == '0100' && op2<0> == '0');
    else
        return FALSE;
```

### Library pseudocode for aarch64/functions/system/AArch64.ExecutingERETInstr

```
// AArch64.ExecutingERETInstr()
// =====
// Returns TRUE if current instruction is ERET.

boolean AArch64.ExecutingERETInstr()
    instr = ThisInstr();
    return instr<31:12> == '11010110100111110000';
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr

```
// AArch64.ImpDefSysInstr()
// =====
// Execute an implementation-defined system instruction with write (source operand).

AArch64.ImpDefSysInstr(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2, integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstr128

```
// AArch64.ImpDefSysInstr128()
// =====
// Execute an implementation-defined system instruction with write (128-bit source operand).

AArch64.ImpDefSysInstr128(integer el, bits(3) op1, bits(4) CRn,
                           bits(4) CRm, bits(3) op2,
                           integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysInstrWithResult

```
// AArch64.ImpDefSysInstrWithResult()
// =====
// Execute an implementation-defined system instruction with read (result operand).

AArch64.ImpDefSysInstrWithResult(integer el, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2);
```

### 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].

AArch64.ImpDefSysRegRead(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                           integer t);
```

### 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].

AArch64.ImpDefSysRegRead128(bits(2) op0, bits(3) op1, bits(4) CRn,
                              bits(4) CRm, bits(3) op2,
                              integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/AArch64.ImpDefSysRegWrite

```
// AArch64.ImpDefSysRegWrite()
// =====
// Write to an implementation-defined System register.

AArch64.ImpDefSysRegWrite(bits(2) op0, bits(3) op1, bits(4) CRn, bits(4) CRm, bits(3) op2,
                           integer t);
```

### 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.

AArch64.ImpDefSysRegWrite128(bits(2) op0, bits(3) op1, bits(4) CRn,
                               bits(4) CRm, bits(3) op2,
                               integer t, integer t2);
```

### Library pseudocode for aarch64/functions/system/AArch64.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    assert GCR_EL1.RRND == '0';
    constant bits(16) lfsr = RGSR_EL1.SEED<15:0>;
    constant bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSR_EL1.SEED<15:0> = top:lfsr<15:1>;
    return top;
```

### Library pseudocode for aarch64/functions/system/AArch64.RandomTag

```
// AArch64.RandomTag()
// =====
// Generate a random Allocation Tag.

bits(4) AArch64.RandomTag()
    bits(4) tag;
    for i = 0 to 3
        tag<i> = AArch64.NextRandomTagBit\(\);
    return tag;
```

### Library pseudocode for aarch64/functions/system/AArch64.SysInstr

```
// AArch64.SysInstr()
// =====
// Execute a system instruction with write (source operand).

AArch64.SysInstr(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### 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].

AArch64.SysInstrWithResult(integer op0, integer op1, integer crn, integer crm, integer op2,
                           integer t);
```

### 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].

AArch64.SysRegRead(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### Library pseudocode for aarch64/functions/system/AArch64.SysRegWrite

```
// AArch64.SysRegWrite()
// =====
// Write to a System register.

AArch64.SysRegWrite(integer op0, integer op1, integer crn, integer crm, integer op2, integer t);
```

### 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.

boolean BTypeCompatible;
```

### 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.

boolean BTypeCompatible_BTI(bits(2) hintcode)
    case hintcode of
        when '00'
            return FALSE;
        when '01'
            return PSTATE.BTYPE != '11';
        when '10'
            return PSTATE.BTYPE != '10';
        when '11'
            return TRUE;
```

### Library pseudocode for aarch64/functions/system/BTypeCompatible\_PACIXSP

```
// BTypeCompatible_PACIXSP()
// =====
// Returns TRUE if PACIASP, PACIBSP instruction is implicit compatible with PSTATE.BTYPE,
// FALSE otherwise.

boolean BTypeCompatible_PACIXSP()
    if PSTATE.BTYPE IN {'01', '10'} then
        return TRUE;
    elsif PSTATE.BTYPE == '11' then
        index = if PSTATE.EL == ELO then 35 else 36;
        return SCTLRL_ELx[]<index> == '0';
    else
        return FALSE;
```

### 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.

bits(2) BTypeNext;
```

## 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 RGSR_EL1 and/or write RGSR_EL1 to an IMPLEMENTATION DEFINED value.
// If it is not capable of writing RGSR_EL1.SEED[15:0] to zero from a previous nonzero
// RGSR_EL1.SEED value, it is IMPLEMENTATION DEFINED whether the randomness is significantly
// impacted if RGSR_EL1.SEED[15:0] is set to zero.

bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

## 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.

boolean InGuardedPage;
```

## 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.

boolean IsHCRXEL2Enabled()
    if !IsFeatureImplemented(FEAT_HCX) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.HXEn == '0' then
        return FALSE;

    return EL2Enabled\(\);
```

## 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.

boolean IsSCTLR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;
    elseif (EL2Enabled\(\) && (!IsHCRXEL2Enabled\(\) || HCRX_EL2.SCTLR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

### 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.

boolean IsSCTLR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_SCTLR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.SCTLR2En == '0' then
        return FALSE;

    return EL2Enabled\(\);
```

### 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.

boolean IsTCR2EL1Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.TCR2En == '0' then
        return FALSE;
    elsif (EL2Enabled\(\) && (!IsHCRXEL2Enabled\(\) || HCRX_EL2.TCR2En == '0')) then
        return FALSE;
    else
        return TRUE;
```

### 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.

boolean IsTCR2EL2Enabled()
    if !IsFeatureImplemented(FEAT_TCR2) then return FALSE;
    if HaveEL\(EL3\) && SCR_EL3.TCR2En == '0' then
        return FALSE;

    return EL2Enabled\(\);
```

### Library pseudocode for aarch64/functions/system/SetBTypeCompatible

```
// SetBTypeCompatible()
// =====
// Sets the value of BTypeCompatible global variable used by BTI

SetBTypeCompatible(boolean x)
    BTypeCompatible = x;
```

### Library pseudocode for aarch64/functions/system/SetBTypeNext

```
// SetBTypeNext()
// =====
// Set the value of BTypeNext global variable used by BTI

SetBTypeNext(bits(2) x)
    BTypeNext = x;
```

### Library pseudocode for aarch64/functions/system/SetInGuardedPage

```
// SetInGuardedPage()
// =====
// Global state updated to denote if memory access is from a guarded page.

SetInGuardedPage(boolean guardedpage)
    InGuardedPage = guardedpage;
```

### Library pseudocode for aarch64/functions/system128/AArch64.SysInstr128

```
// AArch64.SysInstr128()
// =====
// Execute a system instruction with write (2 64-bit source operands).

AArch64.SysInstr128(integer op0, integer op1, integer crn, integer crm,
                    integer op2, integer t, integer t2);
```

### 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].

AArch64.SysRegRead128(integer op0, integer op1, integer crn, integer crm,
                     integer op2, integer t, integer t2);
```

### 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.

AArch64.SysRegWrite128(integer op0, integer op1, integer crn, integer crm,
                      integer op2, integer t, integer t2);
```

## 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.

AArch64.TLBIP_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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(Xt<107:64> : Zeros(12), 64);
    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();

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);
    return;
```



## 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.

AArch64.TLBIP_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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;

    constant bits(2) tg = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num = UInt(Xt<43:39>);
    constant integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    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();

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBIP_RVA(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBIP_RVAA(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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;

    constant bits(2) tg = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num = UInt(Xt<43:39>);
    constant integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIPRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBIP_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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(Xt<107:64> : Zeros(12), 64);
    r.d64 = r.ttl == '00xx';
    r.d128 = TRUE;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBIP_VAA(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(128) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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(Xt<107:64> : Zeros(12), 64);
    r.d64 = r.ttl == '00xx';
    r.d128 = TRUE;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Broadcast broadcast,
                 TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid,
                  Broadcast broadcast, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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(Xt<39:0> : Zeros(12), 64);
    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 security state does not have stage 2 translation
            Unreachable();

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_PAALL(Broadcast broadcast)
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    TLBIRecord r;

    // 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);

    return;
```

## 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.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
    Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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';

    constant bits(2) tg = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num = UInt(Xt<43:39>);
    constant integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    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();

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```





```

// 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
//
AArch64.TLBI_RPA(TLBILevel level, bits(64) Xt, Broadcast broadcast)
    assert IsFeatureImplemented(FEAT_RME) && PSTATE.EL == EL3;

    TLBIRecord r;
    AddressSize range_bits;
    AddressSize p;

    // 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

    // 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

    if range_bits < p then
        range_bits = p;

    bits(52) BaseADDR = Zeros(52);
    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

    // 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.
    constant integer range_pbits = range_bits;
    constant bits(52) start_addr = BaseADDR AND NOT ZeroExtend(Ones(range_pbits), 52);
    constant bits(52) end_addr = start_addr + ZeroExtend(Ones(range_pbits), 52);

    // PASpace is not considered in TLBI by PA operations
    r.address = ZeroExtend(start_addr, 64);
    r.end_address = ZeroExtend(end_addr, 64);

    TLBI(r);
    if broadcast != Broadcast_NSH then BroadcastTLBI(broadcast, r);

```

## 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.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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';

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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';

    constant bits(2) tg = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num = UInt(Xt<43:39>);
    constant integer baseaddr = SInt(Xt<36:0>);

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
               Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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(Xt<43:0> : Zeros(12), 64);
    r.d64 = TRUE;
    r.d128 = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                Broadcast broadcast, TLBILevel level, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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(Xt<43:0> : Zeros(12), 64);
    r.d64 = TRUE;
    r.d128 = r.ttl == '00xx';

    TLBI(r);
    if broadcast != Broadcast\_NSH then BroadcastTLBI(broadcast, r);
    return;
```

## 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.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                   Broadcast broadcast, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2, EL1};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                      Broadcast broadcast, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};

    TLBIRecord r;
    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);
    return;
```

## 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.

AArch64.TLBI_VMALLWS2(SecurityState security, Regime regime, bits(16) vmid,
                     Broadcast broadcast, TLBIMemAttr attr, bits(64) Xt)
    assert PSTATE.EL IN {EL3, EL2};
    assert regime == Regime\_EL10;

    if security == SS\_Secure && HaveEL(EL3) && SCR_EL3.EEL2 == '0' then
        return;

    TLBIRecord r;
    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);
    return;
```

## Library pseudocode for aarch64/functions/tlbi/ASID\_NONE

```
constant bits(16) ASID_NONE = Zeros(16);
```

## Library pseudocode for aarch64/functions/tlbi/Broadcast

```
// Broadcast
// =====

enumeration Broadcast {
    Broadcast_ISH,
    Broadcast_ForcedISH,
    Broadcast_OSH,
    Broadcast_NSH
};
```

## Library pseudocode for aarch64/functions/tlbi/BroadcastTLBI

```
// BroadcastTLBI()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated broadcast
// domain.

BroadcastTLBI(Broadcast broadcast, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for aarch64/functions/tlbi/DecodeTLBITG

```
// DecodeTLBITG()
// =====
// Decode translation granule size in TLBI range instructions

TGx DecodeTLBITG(bits(2) tg)
    case tg of
        when '01' return TGx\_4KB;
        when '10' return TGx\_16KB;
        when '11' return TGx\_64KB;
```

## Library pseudocode for aarch64/functions/tlbi/GPTTLBIMatch

```
// GPTTLBIMatch()
// =====
// Determine whether the GPT TLB entry lies within the scope of invalidation

boolean GPTTLBIMatch(TLBIRecord tlbi, GPTEntry gpt_entry)
    assert tlbi.op IN {TLBIOp\_RPA, TLBIOp\_PAALL};

    boolean match;
    constant bits(64) entry_size_mask      = ZeroExtend(Ones(gpt_entry.size), 64);
    constant bits(64) entry_end_address    = (ZeroExtend(gpt_entry.pa<55:0> OR
        entry_size_mask<55:0>, 64));

    constant bits(64) entry_start_address = (ZeroExtend(gpt_entry.pa<55:0> AND NOT
        entry_size_mask<55:0>, 64));

    case tlbi.op of
        when TLBIOp\_RPA
            match = (UInt(tlbi.address<55:0>) <= UInt(entry_end_address<55:0>) &&
                UInt(tlbi.end_address<55:0>) > UInt(entry_start_address<55:0>) &&
                (tlbi.level == TLBILevel\_Any || gpt_entry.level == 1));
        when TLBIOp\_PAALL
            match = TRUE;

    return match;
```

## Library pseudocode for aarch64/functions/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
    if !IsFeatureImplemented(FEAT_LPA2) then
        return FALSE;
    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();
```

## 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

boolean ResTLBIRTTL(bits(2) tg, bits(2) ttl)
    case ttl of
        when '00' return TRUE;
        when '01' return DecodeTLBITG(tg) == TGx\_16KB && !IsFeatureImplemented(FEAT_LPA2);
        otherwise return FALSE;
```

## 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

boolean ResTLBITTL(bits(4) ttl)
    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;
```

## Library pseudocode for aarch64/functions/tlbi/TGBits

```
// TGBits()
// =====
// Return the number of least-significant address bits within a single Translation Granule.

AddressSize TGBits(bits(2) tg)
    case tg of
        when '01' return 12; // 4KB
        when '10' return 14; // 16KB
        when '11' return 16; // 64KB
        otherwise
            Unreachable();
```

## Library pseudocode for aarch64/functions/tlbi/TLBI

```
// TLBI()
// =====
// Invalidates TLB entries for which TLBIMatch() returns TRUE.

TLBI(TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for aarch64/functions/tlbi/TLBILevel

```
// TLBILevel
// =====

enumeration TLBILevel {
    TLBILevel_Any,          // this applies to TLB entries at all levels
    TLBILevel_Last         // this applies to TLB entries at last level only
};
```





```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord tlb_entry)
    boolean match;
    constant bits(64) entry_block_mask = ZeroExtend(Ones(tlb_entry.blocksize), 64);
    bits(64) entry_end_address = tlb_entry.context.ia OR entry_block_mask;
    bits(64) entry_start_address = 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);
        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
            boolean regime_match;
            boolean context_match;
            boolean address_match;
            boolean level_match;
            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'));
            constant integer addr_lsb = tlb_entry.blocksize;
            address_match = tlb_entry.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb>;
            level_match = (tlb_entry.level == TLBIlevel\_Any || !tlb_entry.walkstate.istable);
            match = regime_match && context_match && address_match && level_match;
        when TLBIOp\_ALL
            relax_regime = (tlb_entry.from_aarch64 &&
                            tlb_entry.regime IN {Regime\_EL20, Regime\_EL2} &&
                            tlb_entry.context.regime IN {Regime\_EL20, Regime\_EL2});
            match = (tlb_entry.security == tlb_entry.context.ss &&
                    (tlb_entry.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
            constant integer addr_lsb = 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 &&
                    tlb_entry.address<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
                    (!tlb_entry.from_aarch64 || ResTLBITTL(tlb_entry.ttl) || (
                        DecodeTLBITG(tlb_entry.ttl<3:2>) == tlb_entry.context.tg &&
                        UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level)
                    ) &&
                    ((tlb_entry.d128 && tlb_entry.context.isd128) ||
                     (tlb_entry.d64 && !tlb_entry.context.isd128) ||
                     (tlb_entry.d64 && tlb_entry.d128)) &&
                    (tlb_entry.level == TLBIlevel\_Any || !tlb_entry.walkstate.istable));
        when TLBIOp\_VAA, TLBIOp\_VAA
            constant integer addr_lsb = tlb_entry.blocksize;
            match = (tlb_entry.context.includes_s1 &&

```

```

    tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
    tlb_entry.context.ia<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
    (!tlb_entry.from_aarch64 || ResTLBITTL(tlb_entry.ttl) || (
        DecodeTLBITG(tlb_entry.ttl<3:2>) == tlb_entry.context.tg &&
        UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level)
    ) &&
    ((tlb_entry.d128 && tlb_entry.context.isd128) ||
    (tlb_entry.d64 && !tlb_entry.context.isd128) ||
    (tlb_entry.d64 && tlb_entry.d128)) &&
    (tlb_entry.level == TLBILevel Any || !tlb_entry.walkstate.istable));
when TLBIOp VA, TLBIOp VA
    constant integer addr_lsb = tlb_entry.blocksize;
    match = (tlb_entry.context.includes_s1 &&
    tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    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') &&
    tlb_entry.context.ia<55:addr_lsb> == tlb_entry.context.ia<55:addr_lsb> &&
    (!tlb_entry.from_aarch64 || ResTLBITTL(tlb_entry.ttl) || (
        DecodeTLBITG(tlb_entry.ttl<3:2>) == tlb_entry.context.tg &&
        UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level)
    ) &&
    ((tlb_entry.d128 && tlb_entry.context.isd128) ||
    (tlb_entry.d64 && !tlb_entry.context.isd128) ||
    (tlb_entry.d64 && tlb_entry.d128)) &&
    (tlb_entry.level == TLBILevel Any || !tlb_entry.walkstate.istable));
when TLBIOp VMALL
    match = (tlb_entry.context.includes_s1 &&
    tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid));
when TLBIOp VMALLS12
    match = (tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid));
when TLBIOp RIPAS2, TLBIOp RIPAS2
    match = (!tlb_entry.context.includes_s1 && tlb_entry.context.includes_s2 &&
    tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
    tlb_entry.ipaspace == tlb_entry.context.ipaspace &&
    (tlb_entry.tg != '00' && DecodeTLBITG(tlb_entry.tg) == tlb_entry.context.tg) &&
    (!tlb_entry.from_aarch64 || ResTLBITTL(tlb_entry.tg, tlb_entry.ttl<1:0>) ||
    UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level) &&
    ((tlb_entry.d128 && tlb_entry.context.isd128) ||
    (tlb_entry.d64 && !tlb_entry.context.isd128) ||
    (tlb_entry.d64 && tlb_entry.d128)) &&
    UInt(tlb_entry.address<55:0>) <= UInt(entry_end_address<55:0>) &&
    UInt(tlb_entry.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp RVAA, TLBIOp RVAA
    match = (tlb_entry.context.includes_s1 &&
    tlb_entry.context.ss &&
    tlb_entry.context.regime &&
    tlb_entry.context.use_vmid &&
    (!tlb_entry.context.use_vmid || tlb_entry.vmid == tlb_entry.context.vmid) &&
    (tlb_entry.tg != '00' && DecodeTLBITG(tlb_entry.tg) == tlb_entry.context.tg) &&
    (!tlb_entry.from_aarch64 || ResTLBITTL(tlb_entry.tg, tlb_entry.ttl<1:0>) ||
    UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level) &&
    ((tlb_entry.d128 && tlb_entry.context.isd128) ||
    (tlb_entry.d64 && !tlb_entry.context.isd128) ||
    (tlb_entry.d64 && tlb_entry.d128)) &&
    UInt(tlb_entry.address<55:0>) <= UInt(entry_end_address<55:0>) &&
    UInt(tlb_entry.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp RVA, TLBIOp RVA

```

```

    match = (tlb_entry.context.includes_s1 &&
        tlb_entry.context.ss == tlb_entry.context.ss &&
        tlb_entry.context.regime == tlb_entry.context.regime &&
        tlb_entry.context.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') &&
        (tlb_entry.tg != '00' && DecodeTLBITG(tlb_entry.tg) == tlb_entry.context.tg) &&
        (!tlb_entry.from_aarch64 || ResTLBIRTTL(tlb_entry.tg, tlb_entry.ttl<1:0>) ||
            UInt(tlb_entry.ttl<1:0>) == tlb_entry.walkstate.level) &&
        ((tlb_entry.d128 && tlb_entry.context.isd128) ||
            (tlb_entry.d64 && !tlb_entry.context.isd128) ||
            (tlb_entry.d64 && tlb_entry.d128)) &&
        UInt(tlb_entry.address<55:0>) <= UInt(entry_end_address<55:0>) &&
        UInt(tlb_entry.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp\_RPA
    entry_end_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> OR
        entry_block_mask<55:0>);
    entry_start_address<55:0> = (tlb_entry.walkstate.baseaddress.address<55:0> AND
        NOT entry_block_mask<55:0>);
    match = (tlb_entry.context.includes_gpt &&
        UInt(tlb_entry.address<55:0>) <= UInt(entry_end_address<55:0>) &&
        UInt(tlb_entry.end_address<55:0>) > UInt(entry_start_address<55:0>));
when TLBIOp\_PAALL
    match = tlb_entry.context.includes_gpt;

if tlb_entry.attr == TLBI\_ExcludeXS && tlb_entry.context.xs == '1' then
    match = FALSE;

return match;

```

## Library pseudocode for aarch64/functions/tlbi/TLBIMemAttr

```

// TLBIMemAttr
// =====
// Defines the attributes of the memory operations that must be completed in
// order to deem the TLBI operation as completed.

enumeration TLBIMemAttr {
    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 aarch64/functions/tlbi/TLBIOp

```
// TLBIOp
// =====

enumeration TLBIOp {
    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 aarch64/functions/tlbi/TLBIPRange

```
// TLBIPRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIPRange(Regime regime, bits(128) Xt)
    constant boolean valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    constant bits(2) tg     = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num   = UInt(Xt<43:39>);

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    constant AddressSize tg_bits = 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)>;

    constant integer range = (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(start_address<55>, 64-55) : Ones(55);

    return (valid, tg, start_address, end_address);
```

## Library pseudocode for aarch64/functions/tlbi/TLBIRange

```
// TLBIRange()
// =====
// Extract the input address range information from encoded Xt.

(boolean, bits(2), bits(64), bits(64)) TLBIRange(Regime regime, bits(64) Xt)
    constant boolean valid = TRUE;
    bits(64) start_address = Zeros(64);
    bits(64) end_address   = Zeros(64);

    constant bits(2) tg      = Xt<47:46>;
    constant integer scale = UInt(Xt<45:44>);
    constant integer num    = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start_address, end_address);

    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(Xt<36>, 11);
            else
                start_address<48:12> = Xt<36:0>;
                start_address<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start_address<52:16> = Xt<36:0>;
                start_address<63:53> = Replicate(Xt<36>, 11);
            else
                start_address<50:14> = Xt<36:0>;
                start_address<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start_address<52:16> = Xt<36:0>;
            start_address<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    constant integer range = (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(start_address<56>, 64-56) : Ones(56);
    elsif end_address<52> != start_address<52> then
        // overflow, saturate it
        end_address = Replicate(start_address<52>, 64-52) : Ones(52);

    return (valid, tg, start_address, end_address);
```

## Library pseudocode for aarch64/functions/tlbi/TLBIRecord

```
// TLBIRecord
// =====
// Details related to a TLBI operation.

type TLBIRecord is (
    TLBIOp          op,
    boolean          from_aarch64, // originated as an AArch64 operation
    SecurityState    security,
    Regime           regime,
    boolean          use_vmid,
    bits(16)         vmid,
    bits(16)         asid,
    TLBILevel        level,
    TLBIMemAttr      attr,
    PASpace          ipaspace,      // For operations that take IPA as input address
    bits(64)         address,      // input address, for range operations, start address
    bits(64)         end_address,  // for range operations, end address
    boolean          d64,          // For operations that evict VMSAv8-64 based TLB entries
    boolean          d128,        // For operations that evict VMSAv9-128 based TLB entries
    bits(4)          ttl,          // 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
    bits(2)          tg            // for range operations, translation granule
)
```

## Library pseudocode for aarch64/functions/tlbi/VMID

```
// VMID[]
// =====
// Effective VMID.

bits(16) VMID[]
if EL2Enabled() then
    if !ELUsingAArch32(EL2) then
        if IsFeatureImplemented(FEAT_VMID16) && VTCR_EL2.VS == '1' then
            return VTTBR_EL2.VMID;
        else
            return ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        return ZeroExtend(VTTBR.VMID, 16);
elsif HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
    return Zeros(16);
else
    return VMID_NONE;
```

## Library pseudocode for aarch64/functions/tlbi/VMID\_NONE

```
constant bits(16) VMID_NONE = Zeros(16);
```

## Library pseudocode for aarch64/functions/tme/CheckTransactionalSystemAccess

```
// CheckTransactionalSystemAccess()
// =====
// Returns TRUE if an AArch64 MSR, MRS, or SYS instruction is permitted in
// Transactional state, based on the opcode's encoding, and FALSE otherwise.

boolean CheckTransactionalSystemAccess(integer op0, integer op1, integer crn, integer crm,
                                      integer op2, integer read)
    case read<0>:op0<1:0>:op1<2:0>:crn<3:0>:crm<3:0>:op2<2:0> of
        when '0 00 011 0100 xxxx 11x' return TRUE;          // MSR (imm): DAIFSet, DAIFClr
        when '0 01 011 0111 0100 001' return TRUE;          // DC ZVA
        when '0 11 011 0100 0010 00x' return TRUE;          // MSR: NZCV, DAIF
        when '0 11 011 0100 0100 00x' return TRUE;          // MSR: FPCR, FPSR
        when '0 11 000 0100 0110 000' return TRUE;          // MSR: ICC_PMR_EL1
        when '0 11 011 1001 1100 100' return TRUE;          // MRS: PMSWINC_ELO
        when '1 11 011 0010 0101 001' return TRUE;          // MRS: GCSPR_EL0, at EL0
        return PSTATE.EL == EL0;
    // MRS: GCSPR_EL1 at EL1 OR at EL2 when E2H is '1'
    when '1 11 000 0010 0101 001'
        return PSTATE.EL == EL1 || (PSTATE.EL == EL2 && IsInHost());
    when '1 11 100 0010 0101 001'
        return PSTATE.EL == EL2 && !IsInHost();          // MRS: GCSPR_EL2, at EL2 when E2H is '0'
    when '1 11 110 0010 0101 001'
        return PSTATE.EL == EL3;                          // MRS: GCSPR_EL3, at EL3
    when '0 01 011 0111 0111 000' return TRUE;              // GCSPUSHM
    when '1 01 011 0111 0111 001' return TRUE;              // GCSPOPM
    when '0 01 011 0111 0111 010' return TRUE;              // GCSSS1
    when '1 01 011 0111 0111 011' return TRUE;              // GCSSS2
    when '0 01 000 0111 0111 110' return TRUE;              // GCSPOPX
    when '1 11 101 0010 0101 001' return FALSE;             // MRS: GCSPR_EL12
    when '1 11 000 0010 0101 010' return FALSE;             // MRS: GCSCRE0_EL1
    when '1 11 000 0010 0101 000' return FALSE;             // MRS: GCSCR_EL1
    when '1 11 101 0010 0101 000' return FALSE;             // MRS: GCSCR_EL12
    when '1 11 100 0010 0101 000' return FALSE;             // MRS: GCSCR_EL2
    when '1 11 110 0010 0101 000' return FALSE;             // MRS: GCSCR_EL3
    when '1 11 xxx 0xxx xxxx xxx' return TRUE;              // MRS: op0=3, CRn=0..7
    when '1 11 xxx 100x xxxx xxx' return TRUE;              // MRS: op0=3, CRn=8..9
    when '1 11 xxx 1010 xxxx xxx' return TRUE;              // MRS: op0=3, CRn=10
    when '1 11 000 1100 1x00 010' return TRUE;              // MRS: op0=3, CRn=12 - ICC_HPPIRx_EL1
    when '1 11 000 1100 1011 011' return TRUE;              // MRS: op0=3, CRn=12 - ICC_RPR_EL1
    when '1 11 xxx 1101 xxxx xxx' return TRUE;              // MRS: op0=3, CRn=13
    when '1 11 xxx 1110 xxxx xxx' return TRUE;              // MRS: op0=3, CRn=14
    when '0 01 011 0111 0011 111' return TRUE;              // CPP RCTX
    when '0 01 011 0111 0011 10x' return TRUE;              // CFP RCTX, DVP RCTX
    when 'x 11 xxx 1x11 xxxx xxx' return TRUE;              // MRS: op0=3, CRn=11,15
    return (boolean IMPLEMENTATION_DEFINED
            "Accessibility of registers encoded with op0=0b11 and CRn=0b1x11 is allowed");
    otherwise return FALSE;                                // All other SYS, SYSL, MRS, MSR
```

## Library pseudocode for aarch64/functions/tme/CommitTransactionalWrites

```
// CommitTransactionalWrites()
// =====
// Makes all transactional writes to memory observable by other PEs and reset
// the transactional read and write sets.

CommitTransactionalWrites();
```

## Library pseudocode for aarch64/functions/tme/DiscardTransactionalWrites

```
// DiscardTransactionalWrites()
// =====
// Discards all transactional writes to memory and reset the transactional
// read and write sets.

DiscardTransactionalWrites();
```



## Library pseudocode for aarch64/functions/tme/FailTransaction

```
// FailTransaction()
// =====

FailTransaction(TMFailure cause, boolean retry)
    FailTransaction(cause, retry, FALSE, Zeros(15));
    return;

// FailTransaction()
// =====
// Exits Transactional state and discards transactional updates to registers
// and memory.

FailTransaction(TMFailure cause, boolean retry, boolean interrupt, bits(15) reason)
    assert !retry || !interrupt;

    if IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.EL) then
        BRBFCR_EL1.LASTFAILED = '1';

    DiscardTransactionalWrites();
    // For trivial implementation no transaction checkpoint was taken
    if cause != TMFailure TRIVIAL then
        RestoreTransactionCheckpoint();
    ClearExclusiveLocal(ProcessorID());

    bits(64) result = Zeros(64);

    result<23> = if interrupt then '1' else '0';
    result<15> = if retry && !interrupt then '1' else '0';
    case cause of
        when TMFailure TRIVIAL result<24> = '1';
        when TMFailure DBG result<22> = '1';
        when TMFailure NEST result<21> = '1';
        when TMFailure SIZE result<20> = '1';
        when TMFailure ERR result<19> = '1';
        when TMFailure IMP result<18> = '1';
        when TMFailure MEM result<17> = '1';
        when TMFailure CNCL result<16> = '1'; result<14:0> = reason;

    TSTATE.depth = 0;
    X[TSTATE.Rt, 64] = result;
    constant boolean branch_conditional = FALSE;
    BranchTo(TSTATE.nPC, BranchType TMFAIL, branch_conditional);
    EndOfInstruction();
    return;
```

## Library pseudocode for aarch64/functions/tme/IsTMEEnabled

```
// IsTMEEnabled()
// =====
// Returns TRUE if access to TME instruction is enabled, FALSE otherwise.

boolean IsTMEEnabled()
    if PSTATE.EL IN {EL0, EL1, EL2} && HaveEL(EL3) then
        if SCR_EL3.TME == '0' then
            return FALSE;
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        if HCR_EL2.TME == '0' then
            return FALSE;
    return TRUE;
```

## Library pseudocode for aarch64/functions/tme/MemHasTransactionalAccess

```
// MemHasTransactionalAccess()
// =====
// Function checks if transactional accesses are not supported for an address
// range or memory type.

boolean MemHasTransactionalAccess(MemoryAttributes memattrs)
    if ((memattrs.shareability == Shareability\_ISH ||
        memattrs.shareability == Shareability\_OSH) &&
        memattrs.memtype == MemType\_Normal &&
        memattrs.inner.attrs == MemAttr\_WB &&
        memattrs.inner.hints == MemHint\_RWA &&
        memattrs.inner.transient == FALSE &&
        memattrs.outer.hints == MemHint\_RWA &&
        memattrs.outer.attrs == MemAttr\_WB &&
        memattrs.outer.transient == FALSE) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Memory Region does not support Transactional access";
```

## Library pseudocode for aarch64/functions/tme/RestoreTransactionCheckpoint

```
// RestoreTransactionCheckpoint()
// =====
// Restores part of the PE registers from the transaction checkpoint.

RestoreTransactionCheckpoint()
    SP[64] = TSTATE.SP;
    ICC_PMR_EL1 = TSTATE.ICC_PMR_EL1;
    PSTATE.<N,Z,C,V> = TSTATE.nzcv;
    PSTATE.<D,A,I,F> = TSTATE.<D,A,I,F>;

    for n = 0 to 30
        X[n, 64] = TSTATE.X[n];

    if IsFPEEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant VecLen VL = CurrentVL;
            constant PredLen PL = VL DIV 8;
            for n = 0 to 31
                Z[n, VL] = TSTATE.Z[n]<VL-1:0>;
            for n = 0 to 15
                P[n, PL] = TSTATE.P[n]<PL-1:0>;
                FFR[PL] = TSTATE.FFR<PL-1:0>;
        else
            for n = 0 to 31
                V[n, 128] = TSTATE.Z[n]<127:0>;
            FPCR = TSTATE.FPCR;
            FPSR = TSTATE.FPSR;

    if IsFeatureImplemented(FEAT_GCS) then
        case PSTATE.EL of
            when EL0 GCSPR_EL0 = TSTATE.GCSPR_ELx;
            when EL1 GCSPR_EL1 = TSTATE.GCSPR_ELx;
            when EL2 GCSPR_EL2 = TSTATE.GCSPR_ELx;
            when EL3 GCSPR_EL3 = TSTATE.GCSPR_ELx;

    return;
```

## Library pseudocode for aarch64/functions/tme/StartTrackingTransactionalReadsWrites

```
// StartTrackingTransactionalReadsWrites()
// =====
// Starts tracking transactional reads and writes to memory.

StartTrackingTransactionalReadsWrites();
```

## Library pseudocode for aarch64/functions/tme/TMFailure

```
// TMFailure
// =====
// Transactional failure causes

enumeration TMFailure {
    TMFailure_CNCL,    // Executed a TCANCEL instruction
    TMFailure_DBG,     // A debug event was generated
    TMFailure_ERR,     // A non-permissible operation was attempted
    TMFailure_NEST,    // The maximum transactional nesting level was exceeded
    TMFailure_SIZE,    // The transactional read or write set limit was exceeded
    TMFailure_MEM,     // A transactional conflict occurred
    TMFailure_TRIVIAL, // Only a TRIVIAL version of TM is available
    TMFailure_IMP      // Any other failure cause
};
```

## Library pseudocode for aarch64/functions/tme/TMState

```
// TMState
// =====
// Transactional execution state bits.
// There is no significance to the field order.

type TMState is (
    integer    depth,           // Transaction nesting depth
    integer    Rt,              // TSTART destination register
    bits(64)   nPC,             // Fallback instruction address
    array[0..30] of bits(64) X,  // General purpose registers
    array[0..31] of bits(MAX_VL) Z, // Vector registers
    array[0..15] of bits(MAX_PL) P, // Predicate registers
    bits(MAX_PL) FFR,           // First Fault Register
    bits(64)   SP,              // Stack Pointer at current EL
    bits(64)   FPCR,            // Floating-point Control Register
    bits(64)   FPSR,            // Floating-point Status Register
    bits(64)   ICC_PMR_EL1,     // Interrupt Controller Interrupt Priority Mask Register
    bits(64)   GCSPR_ELx,      // GCS pointer for current EL
    bits(4)    nzcv,           // Condition flags
    bits(1)    D,              // Debug mask bit
    bits(1)    A,              // SError interrupt mask bit
    bits(1)    I,              // IRQ mask bit
    bits(1)    F               // FIQ mask bit
)
```

## Library pseudocode for aarch64/functions/tme/TSTATE

```
// TSTATE
// =====
// Global per-transaction state

TMState TSTATE;
```

## Library pseudocode for aarch64/functions/tme/TakeTransactionCheckpoint

```
// TakeTransactionCheckpoint()
// =====
// Captures part of the PE registers into the transaction checkpoint.

TakeTransactionCheckpoint()
    TSTATE.SP          = SP[64];
    TSTATE.ICC_PMR_EL1 = ICC_PMR_EL1;
    TSTATE.nzcv        = PSTATE.<N,Z,C,V>;
    TSTATE.<D,A,I,F>    = PSTATE.<D,A,I,F>;

    for n = 0 to 30
        TSTATE.X[n] = X[n, 64];

    if IsFPEEnabled(PSTATE.EL) then
        if IsSVEEnabled(PSTATE.EL) then
            constant VecLen VL = CurrentVL;
            constant PredLen PL = VL DIV 8;
            for n = 0 to 31
                TSTATE.Z[n]<VL-1:0> = Z[n, VL];
            for n = 0 to 15
                TSTATE.P[n]<PL-1:0> = P[n, PL];
            TSTATE.FFR<PL-1:0> = FFR[PL];
        else
            for n = 0 to 31
                TSTATE.Z[n]<127:0> = V[n, 128];
            TSTATE.FPCR = FPCR;
            TSTATE.FPSR = FPSR;

    if IsFeatureImplemented(FEAT_GCS) then
        case PSTATE.EL of
            when EL0 TSTATE.GCSPR_ELx = GCSPR_EL0;
            when EL1 TSTATE.GCSPR_ELx = GCSPR_EL1;
            when EL2 TSTATE.GCSPR_ELx = GCSPR_EL2;
            when EL3 TSTATE.GCSPR_ELx = GCSPR_EL3;

    return;
```

## Library pseudocode for aarch64/functions/tme/TransactionStartTrap

```
// TransactionStartTrap()
// =====
// Traps the execution of TSTART instruction.

TransactionStartTrap(integer dreg)
    bits(2) targetEL;
    constant bits(64) preferred_exception_return = ThisInstrAddr(64);
    vect_offset = 0x0;

    except = ExceptionSyndrome(Exception TSTARTAccessTrap);
    except.syndrome.iss<9:5> = dreg<4:0>;

    if UInt(PSTATE.EL) > UInt(EL1) then
        targetEL = PSTATE.EL;
    elsif EL2Enabled() && HCR_EL2.TGE == '1' then
        targetEL = EL2;
    else
        targetEL = EL1;
    AArch64.TakeException(targetEL, except, preferred_exception_return, vect_offset);
```

## Library pseudocode for aarch64/functions/vbitop/VBitOp

```
// VBitOp
// =====
// Vector bit select instruction types.

enumeration VBitOp    {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

bits(8) AArch64.MAIRAttr(integer index, MAIRType mair2, MAIRType mair)
    assert (index < 8 || (IsFeatureImplemented(FEAT_AIE) && (index < 16)));
    if (index > 7) then
        return Elem[mair2, index-8, 8]; // Read from LSB at MAIR2
    else
        return Elem[mair, index, 8];
```

## 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.

FaultRecord AArch64.CheckBreakpoint(FaultRecord fault_in, bits(64) vaddress,
                                     AccessDescriptor accdesc, integer size)

    assert !ELUsingAArch32(S1TranslationRegime());
    assert (UsingAArch32() && size IN {2,4}) || size == 4;

    FaultRecord fault = fault_in;
    boolean match = FALSE;
    boolean addr_match_bp = FALSE; // Default assumption that all address match breakpoints
                                   // are inactive or disabled.
    boolean addr_mismatch_bp = FALSE; // Default assumption that all address mismatch
                                      // breakpoints are inactive or disabled.
    boolean addr_match = FALSE;
    boolean addr_mismatch = TRUE; // Default assumption that the given virtual address is
                                  // outside the range of all address mismatch breakpoints
    boolean ctxt_match = FALSE;

    for i = 0 to NumBreakpointsImplemented() - 1
        constant BreakpointInfo brkptinfo = 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;
    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);

    match = match || ctxt_match;

    if match then
        fault.statuscode = Fault Debug;
        fault.vaddress = vaddress;
        if HaltOnBreakpointOrWatchpoint() then
            reason = DebugHalt Breakpoint;
            Halt(reason);

    return fault;
```

## 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.

FaultRecord AArch64.CheckDebug(bits(64) vaddress, AccessDescriptor accdesc, integer size)

    FaultRecord fault = NoFault(accdesc, vaddress);
    boolean generate_exception;

    constant boolean d_side = (IsDataAccess(accdesc.acctype) || accdesc.acctype == AccessType\_DC);
    constant boolean i_side = (accdesc.acctype == AccessType\_IFETCH);
    if accdesc.acctype == AccessType\_NV2 then
        mask = '0';
        ss = CurrentSecurityState();
        generate_exception = (AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) &&
                               MDSCR_EL1.MDE == '1');
    else
        generate_exception = AArch64.GenerateDebugExceptions() && MDSCR_EL1.MDE == '1';
    halt = 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);

    return fault;
```



```

// 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.

FaultRecord AArch64.CheckWatchpoint(FaultRecord fault_in, bits(64) vaddress_in,
                                     AccessDescriptor accdesc, integer size_in)
    assert !ELUsingAArch32(S1TranslationRegime());
    FaultRecord fault      = fault_in;
    FaultRecord fault_match = fault_in;
    FaultRecord fault_mismatch = fault_in;
    bits(64) vaddress      = vaddress_in;
    integer size           = size_in;
    boolean rounded_match   = FALSE;
    constant bits(64) original_vaddress = vaddress;
    constant integer original_size = size;
    boolean addr_match_wp   = FALSE; // Default assumption that all address match watchpoints
                                   // are inactive or disabled.
    boolean addr_mismatch_wp = FALSE; // Default assumption that all address mismatch
                                   // watchpoints are inactive or disabled.
    boolean addr_match       = FALSE;
    boolean addr_mismatch    = TRUE; // Default assumption that the given virtual address is
                                   // outside the range of all address mismatch watchpoints

    if accdesc.acctype == AccessType\_DC then
        if accdesc.cacheop != CacheOp\_Invalidate then
            return fault;
    elseif !IsDataAccess(accdesc.acctype) then
        return fault;

    // 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
        integer upper_vaddress = UInt(original_vaddress) + original_size;
        if ConstrainUnpredictableBool(Unpredictable\_16BYTEROUNDEDDOWNACCESS) then
            vaddress = Align(vaddress, 16);
            rounded_match = TRUE;
        if ConstrainUnpredictableBool(Unpredictable\_16BYTEROUNDEDUPACCESS) then
            upper_vaddress = Align(upper_vaddress + 15, 16) ;
            rounded_match = TRUE;
        size = upper_vaddress - UInt(vaddress);

    for i = 0 to NumWatchpointsImplemented() - 1
        constant WatchpointInfo watchptinfo = 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;
                elseif DBGWCR_EL1[i].LSC<1> == '1' && accdesc.write then
                    fault_match.write = TRUE;
                fault_match.watchptinfo = watchptinfo;
            elseif 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;
            fault_mismatch.watchptinfo = watchptinfo;
        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;
        fault.vaddress = vaddress;
        fault.watchptinfo.maybe_false_match = rounded_match;
        if (fault.statuscode == Fault\_Debug && HaltOnBreakpointOrWatchpoint() &&
            !accdesc.nonfault && !(accdesc.firstfault && !accdesc.first)) then
            reason = DebugHalt\_Watchpoint;
            EDWAR = fault.vaddress;
            is_async = FALSE;
            Halt(reason, is_async, fault);
        return fault;

```



```

// 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.

FaultRecord AppendToHDBSS(FaultRecord fault_in, FullAddress ipa_in, AccessDescriptor accdesc,
                          S2TTWParams walkparams, integer level)
{
    assert CanAppendToHDBSS();

    FaultRecord fault = fault_in;
    FullAddress ipa = ipa_in;
    constant integer hdbss_size = UInt(HDBSSBR_EL2.SZ);

    AddressDescriptor hdbss_addrdesc;

    bits(56) baddr = HDBSSBR_EL2.BADDR : Zeros(12);
    baddr<11 + hdbss_size : 12> = Zeros(hdbss_size);

    hdbss_addrdesc.paddress.address = baddr + (8 * UInt(HDBSSPROD_EL2.INDEX));
    constant bit nse2 = '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);
    constant AccessDescriptor hdbss_access = 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 (boolean IMPLEMENTATION_DEFINED
            "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;

            return fault;

    // The reported IPA must be aligned to the size of the translation.
    constant AddressSize lsb = TranslationSize(walkparams.d128, walkparams.tgx, level);
    ipa.address = ipa.address<55:lsb> : Zeros(lsb);
    bits(64) hdbss_entry = CreateHDBSSEntry(ipa, hdbss_access.ss, level);

    if walkparams.ee == '1' then
        hdbss_entry = BigEndianReverse(hdbss_entry);

    constant PhysMemRetStatus memstatus = PhysMemWrite(hdbss_addrdesc, 8, hdbss_access,
                                                       hdbss_entry);

    if IsFault(memstatus) then
        if (boolean IMPLEMENTATION_DEFINED
            "External Abort on HDBSS write reported in HDBSSPROD_EL2") then
            HDBSSPROD_EL2.FSC = '010000';
        else
            constant boolean iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, hdbss_addrdesc,
                                           hdbss_access, 8, fault);

            fault.level = level;
            fault.hdbssf = TRUE;
        else
            HDBSSPROD_EL2.INDEX = HDBSSPROD_EL2.INDEX + 1;

    return fault;
}

```

## Library pseudocode for aarch64/translation/hdbss/CanAppendToHDBSS

```
// CanAppendToHDBSS()
// =====
// Return TRUE if HDBSS can be appended.

boolean CanAppendToHDBSS()
    if !IsFeatureImplemented(FEAT_HDBSS) then
        return FALSE;
    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;
```

## Library pseudocode for aarch64/translation/hdbss/CreateHDBSSEntry

```
// CreateHDBSSEntry()
// =====
// Returns a HDBSS entry.

bits(64) CreateHDBSSEntry(FullAddress ipa, SecurityState ss, integer level)
    constant bit ns_ipa = if ss == SS\_Secure && ipa.paspace == PAS\_NonSecure then '1' else '0';
    return ZeroExtend(ipa.address<55:12> : ns_ipa : Zeros(7) : level<2:0> : '1', 64);
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

AddressSize AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.LeafBase

```
// AArch64.LeafBase()
// =====
// Extract the address embedded in a block and page descriptor pointing to the
// base of a memory block

bits(56) AArch64.LeafBase(bits(N) descriptor, bit d128, bit ds,
    TGx txx, integer level)
    bits(56) leafbase = Zeros(56);

    granulebits = TGxGranuleBits(txx);
    descsize2 = if d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsize2;
    constant integer leafsize = granulebits + stride * (FINAL\_LEVEL - level);

    leafbase<47:0> = Align(descriptor<47:0>, 1 << leafsize);

    if d128 == '1' then
        leafbase<55:48> = descriptor<55:48>;
    elseif txx == TGx\_64KB && (AArch64.PAMax() >= 52 ||
        boolean IMPLEMENTATION_DEFINED "descriptor[15:12] for 64KB granule are OA[51:48]") then
        leafbase<51:48> = descriptor<15:12>;
    elseif ds == '1' then
        leafbase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return leafbase;
```

## 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

bits(56) AArch64.NextTableBase(bits(N) descriptor, bit d128, bits(2) skl, bit ds, TGx tgx)
    bits(56) tablebase = Zeros(56);
    constant AddressSize granulebits = TGxGranuleBits(tgx);
    integer tablesize;

    if d128 == '1' then
        constant integer descsize2 = 4;
        constant integer stride = granulebits - descsize2;
        tablesize = stride*(1 + UInt(skl)) + descsize2;
    else
        tablesize = granulebits;

    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>;

    tablebase = Align(tablebase, 1 << tablesize);

    if d128 == '1' then
        tablebase<55:48> = descriptor<55:48>;
    elsif tgx == TGx\_64KB && (AArch64.PAMax() >= 52 ||
        boolean IMPLEMENTATION_DEFINED "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>;

    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

AddressSize AArch64.PhysicalAddressSize(bit d128, bit ds, bits(3) encoded_ps, TGx tgx)
    integer ps;
    integer max_ps;

    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;

    if d128 == '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());

    return Min(ps, max_ps);
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S1SLTTEntryAddress

```
// AArch64.S1SLTTEntryAddress()
// =====
// Compute the first stage 1 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S1SLTTEntryAddress(integer level, S1TTWParams walkparams,
                                         bits(64) ia, FullAddress tablebase)

    // Input Address size
    constant integer descsize2 = if walkparams.d128 == '1' then 4 else 3;
    iasize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride = granulebits - descsize2;
    levels = FINAL\_LEVEL - level;

    bits(56) index;
    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = iasize - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsize2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S1StartLevel

```
// AArch64.S1StartLevel()
// =====
// Compute the initial lookup level when performing a stage 1 translation
// table walk

integer AArch64.S1StartLevel(S1TTWParams walkparams)
    // Input Address size
    iasize = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    constant integer descsize2 = if walkparams.d128 == '1' then 4 else 3;
    constant integer stride = granulebits - descsize2;
    integer slstartlevel = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
    if walkparams.d128 == '1' then
        slstartlevel = slstartlevel + UInt(walkparams.skl);
    return slstartlevel;
```

## 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

bits(56) AArch64.S1TTBaseAddress(S1TTWParams walkparams, Regime regime, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizeLog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    startlevel  = AArch64.S1StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizeLog2;

    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>;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
        (IsFeatureImplemented(FEAT_LPA) ||
        boolean IMPLEMENTATION_DEFINED "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>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S2SLTTEntryAddress

```
// AArch64.S2SLTTEntryAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEntryAddress(S2TTWParams walkparams, bits(56) ipa,
    FullAddress tablebase)
    startlevel  = AArch64.S2StartLevel(walkparams);
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    constant descsizeLog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    levels      = FINAL_LEVEL - startlevel;

    bits(56) index;
    constant AddressSize lsb = levels*stride + granulebits;
    constant AddressSize msb = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(descsizeLog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.S2StartLevel

```
// AArch64.S2StartLevel()
// =====
// Determine the initial lookup level when performing a stage 2 translation
// table walk

integer AArch64.S2StartLevel(S2TTWParams walkparams)
    if walkparams.d128 == '1' then
        iasize      = AArch64.IASize(walkparams.txsz);
        granulebits = TGxGranuleBits(walkparams.tgx);
        descsizelog2 = 4;
        constant integer stride = granulebits - descsizelog2;
        integer s2startlevel = FINAL\_LEVEL - (((iasize-1) - granulebits) DIV stride);
        s2startlevel = s2startlevel + UInt(walkparams.sk1);

        return s2startlevel;

    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;
        when TGx\_16KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
                when '11' return 0;
        when TGx\_64KB
            case walkparams.sl0 of
                when '00' return 3;
                when '01' return 2;
                when '10' return 1;
```



## 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

bits(56) AArch64.S2TTBaseAddress(S2TTWParams walkparams, PAspace paspace, bits(N) ttbr)
    bits(56) tablebase = Zeros(56);

    // Input Address size
    iasize      = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsizeLog2 = if walkparams.d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    startlevel  = AArch64.S2StartLevel(walkparams);
    levels      = FINAL_LEVEL - startlevel;

    // Base address is aligned to size of the initial translation table in bytes
    tsize = (iasize - (levels*stride + granulebits)) + descsizeLog2;

    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>;
    elsif walkparams.ds == '1' || (walkparams.tgx == TGx_64KB && walkparams.ps == '110' &&
        (IsFeatureImplemented(FEAT_LPA) ||
        boolean IMPLEMENTATION_DEFINED "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>;
    tablebase = Align(tablebase, 1 << tsize);
    return tablebase;
```

## Library pseudocode for aarch64/translation/vmsa\_addrcalc/AArch64.TTEntryAddress

```
// AArch64.TTEntryAddress()
// =====
// Compute translation table descriptor address within the table pointed to by
// the table base

FullAddress AArch64.TTEntryAddress(integer level, bit d128, bits(2) skl, TGx tgx, bits(6) txsz,
    bits(64) ia, FullAddress tablebase)

    // Input Address size
    iasize      = AArch64.IASize(txsz);
    granulebits = TGxGranuleBits(tgx);
    constant descsizeLog2 = if d128 == '1' then 4 else 3;
    stride      = granulebits - descsizeLog2;
    levels      = FINAL_LEVEL - level;

    bits(56) index;

    constant AddressSize lsb = levels*stride + granulebits;
    constant integer nstride = if d128 == '1' then UInt(skl) + 1 else 1;
    constant AddressSize msb = (lsb + (stride * nstride)) - 1;
    index = ZeroExtend(ia<msb:lsb>:Zeros(descsizeLog2), 56);

    FullAddress descaddress;
    descaddress.address = tablebase.address OR index;
    descaddress.paspace = tablebase.paspace;

    return descaddress;
```

### 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.

AddressSize AArch64.AddrTop(bit tbid, AccessType acctype, bit tbi)
    if tbid == '1' && acctype == AccessType\_IFETCH then
        return 63;

    if tbi == '1' then
        return 55;
    else
        return 63;
```

### 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

boolean AArch64.ContiguousBitFaults(bit d128, bits(6) txsz, TGx txx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize = TranslationSize(d128, txx, level) + ContiguousSize(d128, txx, level);

    return (tsize > iasize &&
        boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit");
```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(56) ipa, S2TTWParams walkparams)
    //Input Address size
    constant integer iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 56 then
        return !IsZero(ipa<55:iasize>);
    else
        return FALSE;
```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.OAOutOfRange

```
// AArch64.OAOutOfRange()
// =====
// Returns whether output address is expressed in the configured size number of bits

boolean AArch64.OAOutOfRange(bits(56) address, bit d128, bit ds, bits(3) ps, TGx txx)
    // Output Address size
    constant integer oasize = AArch64.PhysicalAddressSize(d128, ds, ps, txx);

    if oasize < 56 then
        return !IsZero(address<55:oasize>);
    else
        return FALSE;
```



```

// AArch64.S1CheckPermissions()
// =====
// Checks whether stage 1 access violates permissions of target memory
// and returns a fault record

FaultRecord AArch64.S1CheckPermissions(FaultRecord fault_in, Regime regime, TTWState walkstate,
                                       S1TTWParams walkparams, AccessDescriptor accdesc)

    FaultRecord fault = fault_in;
    constant Permissions permissions = walkstate.permissions;
    S1AccessControls slperms;

    slperms = AArch64.S1ComputePermissions(regime, walkstate, walkparams, accdesc);

    if accdesc.acctype == AccessType_IFETCH then
        if slperms.overlay && slperms.ox == '0' then
            fault.statuscode = Fault_Permission;
            fault.overlay = TRUE;
        elseif (walkstate.memattrs.memtype == MemType_Device &&
                ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT) then
            fault.statuscode = Fault_Permission;
        elseif slperms.x == '0' then
            fault.statuscode = Fault_Permission;
    elseif 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;
            elseif slperms.w == '0' then
                fault.statuscode = Fault_Permission;
            // DC from privileged context which clean cannot generate a Permission fault
        elseif accdesc.el == EL0 then
            if slperms.overlay && slperms.or == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    slperms.overlay && slperms.ow == '0') then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif slperms.r == '0' then
                fault.statuscode = Fault_Permission;
            elseif (walkparams.cmow == '1' &&
                    accdesc.opscope == CacheOpScope_PoC &&
                    accdesc.cacheop == CacheOp_CleanInvalidate &&
                    slperms.w == '0') then
                fault.statuscode = Fault_Permission;
    elseif accdesc.acctype == AccessType_IC then
        // IC from privileged context cannot generate Permission fault
        if accdesc.el == EL0 then
            if (slperms.overlay && slperms.or == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif walkparams.cmow == '1' && slperms.overlay && slperms.ow == '0' then
                fault.statuscode = Fault_Permission;
                fault.overlay = TRUE;
            elseif (slperms.r == '0' &&
                boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
                fault.statuscode = Fault_Permission;
            elseif walkparams.cmow == '1' && slperms.w == '0' then
                fault.statuscode = Fault_Permission;
    elseif IsFeatureImplemented(FEAT_GCS) && accdesc.acctype == AccessType_GCS then
        if slperms.gcs == '0' then
            fault.statuscode = Fault_Permission;
        elseif accdesc.write && walkparams.<ha,hd> != '11' && permissions.ndirty == '1' then
            fault.statuscode = Fault_Permission;
            fault.dirtybit = TRUE;
            fault.write = TRUE;
        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\_CanonicallyTagged) then
    fault.statuscode = Fault\_Permission;
    fault.write      = TRUE;
    fault.sltagnodata = TRUE;
elseif (accdesc.write && !(walkparams.<ha,hd> == '11') && walkparams.pie == '1' &&
        permissions.ndirty == '1') then
    fault.statuscode = Fault\_Permission;
    fault.dirtybit   = TRUE;
    fault.write      = TRUE;

return fault;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1ComputePermissions

```

// AArch64.S1ComputePermissions()
// =====
// Computes the overall stage 1 permissions

S1AccessControls AArch64.S1ComputePermissions(Regime regime, TTWState walkstate,
                                              S1TTWParams walkparams, AccessDescriptor accdesc)
constant Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

if walkparams.pie == '1' then
    slperms = AArch64.S1IndirectBasePermissions(regime, walkstate, walkparams, accdesc);
else
    slperms = AArch64.S1DirectBasePermissions(regime, walkstate, walkparams, accdesc);

if accdesc.el == EL0 && !AArch64.S1E0POEnabled(regime, walkparams.nv1) then
    slperms.overlay = FALSE;
elseif accdesc.el != EL0 && !AArch64.S1POEnabled(regime) then
    slperms.overlay = FALSE;

if slperms.overlay then
    sloverlay_perms = AArch64.S1OverlayPermissions(regime, walkstate, accdesc);
    slperms.or = sloverlay_perms.or;
    slperms.ow = sloverlay_perms.ow;
    slperms.ox = sloverlay_perms.ox;

if slperms.overlay && slperms.wxn == '1' && slperms.ox == '1' then
    // WXN removes overlay write permission if overlay execute permission is not removed.
    slperms.ow = '0';
elseif slperms.wxn == '1' then
    // In the absence of overlay permissions, if WXN is enabled and both W and X
    // permission are granted, the X permission is removed.
    slperms.x = '0';

return slperms;

```



```

// AArch64.S1DirectBasePermissions()
// =====
// Computes the stage 1 direct base permissions

S1AccessControls AArch64.S1DirectBasePermissions(Regime regime, TTWState walkstate,
S1TTWParams walkparams, AccessDescriptor accdesc)

bit r, w, x;
bit pr, pw, px;
bit ur, uw, ux;
Permissions permissions = walkstate.permissions;
S1AccessControls slperms;
// 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';

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

    // 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

    // 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);

    if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime\_EL10 &&
        walkparams.nv1 == '1')) then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
            accdesc.ss == SS\_Secure &&
            walkstate.baseaddress.paspace == PAS\_NonSecure &&
            walkparams.sif == '1') then
            ux = '0';

        if (boolean IMPLEMENTATION_DEFINED "Realm EL2&0 regime affects EPAN" &&
            accdesc.ss == SS\_Realm && regime == Regime\_EL20 &&
            walkstate.baseaddress.paspace != PAS\_Realm) then
            ux = '0';

        pan = PSTATE.PAN AND (ur OR uw OR (walkparams.epan AND ux));
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    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

        // 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

        px = NOT(permissions.xn OR permissions.xn_table);

    (r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);

// Compute WXN value

```

```

wxn = walkparams.wxn AND w AND x;

// 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);
// Prevent execution from non-Root space by Root
if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
    x = '0';
// 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';

slperms.r    = r;
slperms.w    = w;
slperms.x    = x;
slperms.gcs  = '0';
slperms.wxn  = wxn;
slperms.overlay = TRUE;

return slperms;

```

### 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

boolean AArch64.S1HasAlignmentFaultDueToMemType(Regime regime, AccessDescriptor accdesc,
    boolean aligned, bit ntlsmd,
    MemoryAttributes memattrs)

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;

if memattrs.memtype != MemType\_Device then
    return FALSE;
elseif ((accdesc.acctype == AccessType\_DCZero && accdesc.cachetype == CacheType\_Tag) ||
    accdesc.stzgm) then
    return ConstrainUnpredictable(Unpredictable\_DEVICETAGSTORE) == Constraint\_FAULT;
elseif accdesc.a32lsmd && ntlsmd == '0' then
    return memattrs.device != DeviceType\_GRE;
elseif accdesc.acctype == AccessType\_DCZero then
    return TRUE;
elseif !aligned then
    return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
else
    return FALSE;

```





```

// AArch64.S1IndirectBasePermissions()
// =====
// Computes the stage 1 indirect base permissions

S1AccessControls AArch64.S1IndirectBasePermissions(Regime regime, TTWState walkstate,
S1TTWParams walkparams,
AccessDescriptor accdesc)

bit r, w, x, gcs, wxn, overlay;
bit pr, pw, px, pgcs, pwxn, p_overlay;
bit ur, uw, ux, ugcs, uwxn, u_overlay;
constant Permissions permissions = walkstate.permissions;
S1AccessControls slperms;

// 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','0'); // 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

p_overlay = NOT(permissions.ppi<3>);
pwxn = if permissions.ppi == '0110' then '1' else '0';

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
        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','0'); // 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
        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
        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

    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');

    if (IsFeatureImplemented(FEAT_PAN) && accdesc.pan && !(regime == Regime\_EL10 &&
        walkparams.nv1 == '1')) then
        if PSTATE.PAN == '1' && (permissions.upi != '0000') then
            (pr,pw) = ('0','0');

```

```

if accdesc.el == ELO 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);

// 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);
    gcs = '0';
// Prevent execution from non-Root space by Root
if accdesc.ss == SS\_Root && walkstate.baseaddress.paspace != PAS\_Root then
    x = '0';
    gcs = '0';
// 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';
    gcs = '0';

slperms.r      = r;
slperms.w      = w;
slperms.x      = x;
slperms.gcs    = gcs;
slperms.wxn    = wxn;
slperms.overlay = overlay == '1';

return slperms;

```

### Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1OAOutOfRange

```

// AArch64.S1OAOutOfRange()
// =====
// Returns whether stage 1 output address is expressed in the configured size number of bits

boolean AArch64.S1OAOutOfRange(bits(56) address, S1TTWParams walkparams)
    return AArch64.OAOutOfRange(address, walkparams.dl28, walkparams.ds, walkparams.ps,
                                walkparams.tgx);

```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1OverlayPermissions

```
// AArch64.S1OverlayPermissions()
// =====
// Computes the stage 1 overlay permissions

S1AccessControls AArch64.S1OverlayPermissions(Regime regime, TTWState walkstate,
                                              AccessDescriptor accdesc)

bit r, w, x;
bit pr, pw, px;
bit ur, uw, ux;
constant Permissions permissions = walkstate.permissions;
S1AccessControls sloverlay_perms;

constant S1PORType por = AArch64.S1POR(regime);
constant integer bit_index = 4 * UInt(permissions.po_index);
constant bits(4) ppo = 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

if HasUnprivileged(regime) then
  constant bits(4) upo = POR_EL0<bit_index+3:bit_index>;

  // 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

(r,w,x) = if accdesc.el == ELO then (ur,uw,ux) else (pr,pw,px);

sloverlay_perms.or = r;
sloverlay_perms.ow = w;
sloverlay_perms.ox = x;

return sloverlay_perms;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S1TxSZFaults

```
// AArch64.S1TxSZFaults()
// =====
// Detect whether configuration of stage 1 TxSZ field generates a fault

boolean AArch64.S1TxSZFaults(Regime regime, S1TTWParams walkparams)
    mintxs = AArch64.S1MinTxSZ(regime, walkparams.d128, walkparams.ds, walkparams.tgx);
    maxtxs = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txs) < mintxs then
        return (IsFeatureImplemented(FEAT_LVA) ||
            boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txs) > maxtxs then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```



```

// AArch64.S2CheckPermissions()
// =====
// Verifies memory access with available permissions.

(FaultRecord, boolean) AArch64.S2CheckPermissions(FaultRecord fault_in, TTWState walkstate,
S2TTWParams walkparams, AddressDescriptor ipa,
AccessDescriptor accdesc)

constant MemType memtype = walkstate.memattrs.memtype;
constant Permissions permissions = walkstate.permissions;
FaultRecord fault = fault_in;
constant S2AccessControls s2perms = AArch64.S2ComputePermissions(permissions, walkparams,
                                                                    accdesc);

bit r, w;
bit or, ow;

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;

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.hd != '1' || (walkparams.hdbss == '1' && !CanAppendToHDBSS())) &&
        walkparams.s2pie == '1' && permissions.s2dirty == '0') then
        fault.statuscode = Fault\_Permission;
        fault.dirtybit = TRUE;

```

```

        fault.hdbssf = walkparams.hdbss == '1' && !CanAppendToHDBSS();

// 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;

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;
        if !ELUsingAArch32(EL1) && w == '0' then
            fault.statuscode = Fault\_Permission;
    elseif !ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (walkparams.cmow == '1' &&
            accdesc.opscope == CacheOpScope\_PoC &&
            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.cmow == '1' &&
            accdesc.opscope == CacheOpScope\_PoC &&
            accdesc.cacheop == CacheOp\_CleanInvalidate &&
            w == '0') then
        fault.statuscode = Fault\_Permission;

elseif accdesc.acctype == AccessType\_IC then
    if (!ELUsingAArch32(EL1) && accdesc.el == EL0 && s2perms.overlay && or == '0' &&
        boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif walkparams.cmow == '1' && s2perms.overlay && ow == '0' then
        fault.statuscode = Fault\_Permission;
        fault.overlay = TRUE;
    elseif (!ELUsingAArch32(EL1) && accdesc.el == EL0 && r == '0' &&
        boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution") then
        fault.statuscode = Fault\_Permission;
    elseif walkparams.cmow == '1' && w == '0' then
        fault.statuscode = Fault\_Permission;

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) != TCFType\_Ignore) ||
         accdesc.tagaccess) &&
         ipa.memattrs.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();
// MRO* allows only RCW and MMU writes
boolean mro;
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';

return (fault, mro);

```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2ComputePermissions

```

// AArch64.S2ComputePermissions()
// =====
// Compute the overall stage 2 permissions.

S2AccessControls AArch64.S2ComputePermissions(Permissions permissions, S2TTWParams walkparams,
                                              AccessDescriptor accdesc)

    S2AccessControls s2perms;

    if walkparams.s2pie == '1' then
        s2perms = AArch64.S2IndirectBasePermissions(permissions, accdesc);
        s2perms.overlay = IsFeatureImplemented(FEAT_S2POE) && VTCR_EL2.S2POE == '1';
        if s2perms.overlay then
            s2overlay_perms = 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';
        else
            s2perms = AArch64.S2DirectBasePermissions(permissions, accdesc, walkparams);

    return s2perms;

```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2DirectBasePermissions

```
// AArch64.S2DirectBasePermissions()
// =====
// Computes the stage 2 direct base permissions.

S2AccessControls AArch64.S2DirectBasePermissions(Permissions permissions,
AccessDescriptor accdesc, S2TTWParams walkparams)

    S2AccessControls s2perms;
    bit w;
    constant bit r = 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.
    elseif 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';
    else
        w = '0';

    bit px, ux;
    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');

    x = if accdesc.el == EL0 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;
```

## 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

boolean AArch64.S2HasAlignmentFaultDueToMemType(AccessDescriptor accdesc, boolean aligned,
MemoryAttributes memattrs)

    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;

    if memattrs.memtype != MemType\_Device then
        return FALSE;
    elseif ((accdesc.acctype == AccessType\_DCZero && accdesc.cachetype == CacheType\_Tag) ||
        accdesc.stzgm) then
        return ConstrainUnpredictable(Unpredictable DEVICETAGSTORE) == Constraint\_FAULT;
    elseif accdesc.acctype == AccessType\_DCZero then
        return TRUE;
    elseif !aligned then
        return !(boolean IMPLEMENTATION_DEFINED "Device location supports unaligned access");
    else
        return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2InconsistentSL

```
// AArch64.S2InconsistentSL()
// =====
// Detect inconsistent configuration of stage 2 TxSZ and SL fields

boolean AArch64.S2InconsistentSL(S2TTWParams walkparams)
    startlevel = AArch64.S2StartLevel(walkparams);
    levels = FINAL\_LEVEL - startlevel;
    granulebits = TGxGranuleBits(walkparams.tgx);
    descsize = 1 << 3;
    stride = granulebits - descsize;

    // Input address size must at least be large enough to be resolved from the start level
    sl_min_iasize = (
        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
    sl_max_iasize = sl_min_iasize + (stride-1) + 4;
    // Configured Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2IndirectBasePermissions

```
// AArch64.S2IndirectBasePermissions()
// =====
// Computes the stage 2 indirect base permissions.

S2AccessControls AArch64.S2IndirectBasePermissions(Permissions permissions,
AccessDescriptor accdesc)

    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2perms;

    constant bits(4) s2pi = 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','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

    x = 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 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');

    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;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2InvalidSL

```
// AArch64.S2InvalidSL()
// =====
// Detect invalid configuration of SL field

boolean AArch64.S2InvalidSL(S2TTWParams walkparams)
    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;
        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;
        when TGx\_64KB
            case walkparams.sl0 of
                when '11' return TRUE;
                when '10' return AArch64.PAMax() < 44;
                otherwise return FALSE;
```

## 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

boolean AArch64.S2OAOOutOfRange(bits(56) address, S2TTWParams walkparams)
    return AArch64.OAOOutOfRange(address, walkparams.d128, walkparams.ds, walkparams.ps,
                                   walkparams.tgx);
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2OverlayPermissions

```
// AArch64.S2OverlayPermissions()
// =====
// Computes the stage 2 overlay permissions.

S2AccessControls AArch64.S2OverlayPermissions(Permissions permissions, AccessDescriptor accdesc)
    bit r, w;
    bit r_rcw, w_rcw;
    bit r_mmu, w_mmu;
    bit px, ux;
    bit toplevel0, toplevel1;
    S2AccessControls s2overlay_perms;

    constant integer index = 4 * UInt(permissions.s2po_index);
    constant bits(4) s2po = S2POR_EL1<index+3 : index>;
    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

    x = 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');

    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;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.S2TxSZFaults

```
// AArch64.S2TxSZFaults()
// =====
// Detect whether configuration of stage 2 TxSZ field generates a fault

boolean AArch64.S2TxSZFaults(S2TTWParams walkparams, boolean slaarch64)
    mintxs = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds, walkparams.tgx, slaarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    if UInt(walkparams.txsz) < mintxs then
        return (IsFeatureImplemented(FEAT_LPA) ||
            boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum");
    if UInt(walkparams.txsz) > maxtxsz then
        return boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum";

    return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_faults/AArch64.VAIsOutOfRange

```
// AArch64.VAIsOutOfRange()
// =====
// Check bits not resolved by translation are identical and of accepted value

boolean AArch64.VAIsOutOfRange(bits(64) va_in, AccessType acctype,
                                Regime regime, S1TTWParams walkparams)
    bits(64) va = va_in;

    constant AddressSize addrtop = 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';

    // Input Address size
    constant integer iasize = 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;

    if HasUnprivileged(regime) then
        if AArch64.GetVARange(va) == VARange\_LOWER then
            return IsZero(va<addrtop:iasize>);
        else
            return IsOnes(va<addrtop:iasize>);
    else
        return IsZero(va<addrtop:iasize>);
```





```

// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs, S2TTWParams walkparams,
                                             bits(N) descriptor)

    MemoryAttributes memattrs;
    s2_attr = descriptor<5:2>;
    s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    s2_fnxs = descriptor<11>;

    if s2_attr<2> == '0' then          // S2 Device, S1 any
        s2_device = 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;

        memattrs.xs = s1_memattrs.xs;

    elseif s2_attr<1:0> == '11' then    // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elseif 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;

        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;

        memattrs.xs = '0';

    else                                // Non-cacheable unless S1 is device
        if s1_memattrs.memtype == MemType Device then
            memattrs = s1_memattrs;
        else
            MemAttrHints cacheability_attr;
            cacheability_attr.attrs = MemAttr NC;

            memattrs.memtype = MemType Normal;
            memattrs.inner   = cacheability_attr;
            memattrs.outer   = cacheability_attr;

            memattrs.xs = s1_memattrs.xs;

    s2_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';

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;

```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.GetS1TLBContext

```
// AArch64.GetS1TLBContext()
// =====
// Gather translation context for accesses with VA to match against TLB entries

TLBContext AArch64.GetS1TLBContext(Regime regime, SecurityState ss, bits(64) va, TGx tg)
    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();

    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;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.GetS2TLBContext

```
// AArch64.GetS2TLBContext()
// =====
// Gather translation context for accesses with IPA to match against TLB entries

TLBContext AArch64.GetS2TLBContext(SecurityState ss, FullAddress ipa, TGx tg)
    assert EL2Enabled();

    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime  = Regime\_EL10;
    tlbcontext.ipaspace = ipa.paspace;
    tlbcontext.vmid    = VMID[];
    tlbcontext.tg      = tg;
    tlbcontext.ia      = ZeroExtend(ipa.address, 64);
    if IsFeatureImplemented(FEAT_TTCNP) then
        tlbcontext.cnp = if ipa.paspace == PAS\_Secure then VSTTBR_EL2.CnP else VTTBR_EL2.CnP;
    else
        tlbcontext.cnp = '0';

    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;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL10

```
// AArch64.TLBContextEL10()
// =====
// Gather translation context for accesses under EL10 regime to match against TLB entries

TLBContext AArch64.TLBContextEL10(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL10;
    tlbcontext.vmid    = VMID[];

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL1Enabled() && TCR2_EL1.A2 == '1' then
        constant 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;

    if TCR_EL1.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    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;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL2

```
// AArch64.TLBContextEL2()
// =====
// Gather translation context for accesses under EL2 regime to match against TLB entries

TLBContext AArch64.TLBContextEL2(SecurityState ss, bits(64) va, TGx tg)
    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;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL20

```
// AArch64.TLBContextEL20()
// =====
// Gather translation context for accesses under EL20 regime to match against TLB entries

TLBContext AArch64.TLBContextEL20(SecurityState ss, bits(64) va, TGx tg)
    TLBContext tlbcontext;

    tlbcontext.ss      = ss;
    tlbcontext.regime = Regime\_EL20;

    if IsFeatureImplemented(FEAT_ASID2) && IsTCR2EL2Enabled() && TCR2_EL2.A2 == '1' then
        constant 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;

    if TCR_EL2.AS == '0' then
        tlbcontext.asid<15:8> = Zeros(8);
    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;
    else
        tlbcontext.cnp = '0';

    return tlbcontext;
```

## Library pseudocode for aarch64/translation/vmsa\_tlbcontext/AArch64.TLBContextEL3

```
// AArch64.TLBContextEL3()
// =====
// Gather translation context for accesses under EL3 regime to match against TLB entries

TLBContext AArch64.TLBContextEL3(SecurityState ss, bits(64) va, TGx tg)
    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;
```

## 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

AddressDescriptor AArch64.FullTranslate(bits(64) va, AccessDescriptor accdesc, boolean aligned)
    constant Regime regime = TranslationRegime(accdesc.el);
    FaultRecord fault = NoFault(accdesc, va);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, va, aligned, accdesc);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(va, fault);

    if accdesc.ss == SS\_Realm then
        assert EL2Enabled();
    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, slaarch64, aligned, accdesc);

        if fault.statuscode != Fault\_None then
            return CreateFaultyAddressDescriptor(va, fault);
        else
            return pa;
    else
        return ipa;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.MemSwapTableDesc

```
// AArch64.MemSwapTableDesc()
// =====
// Perform HW update of table descriptor as an atomic operation

(FaultRecord, bits(N)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(N) prev_desc,
                                                bits(N) new_desc, bit ee,
                                                AccessDescriptor descaccess,
                                                AddressDescriptor descpaddr)

assert descaccess.acctype == AccessType\_TTW;
FaultRecord fault = fault_in;
boolean iswrite;

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, bits(N) UNKNOWN);

// All observers in the shareability domain observe the
// following memory read and write accesses atomically.
bits(N) mem_desc;
PhysMemRetStatus memstatus;
(memstatus, mem_desc) = PhysMemRead(descpaddr, N DIV 8, descaccess);

if ee == '1' then
    mem_desc = BigEndianReverse(mem_desc);

if IsFault(memstatus) then
    iswrite = FALSE;
    fault = HandleExternalTTWAbort(memstatus, iswrite, descpaddr, descaccess, N DIV 8, fault);
    if IsFault(fault.statuscode) then
        return (fault, bits(N) UNKNOWN);

if mem_desc == prev_desc then
    ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
    memstatus = PhysMemWrite(descpaddr, N DIV 8, 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, bits(N) UNKNOWN);

// Reflect what is now in memory (in little endian format)
mem_desc = new_desc;

return (fault, mem_desc);
```



```

// AArch64.S1DisabledOutput()
// =====
// Map the VA to IPA/PA and assign default memory attributes

(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                         bits(64) va_in, AccessDescriptor accdesc,
                                                         boolean aligned)

bits(64) va = va_in;
walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);
FaultRecord fault = fault_in;

// No memory page is guarded when stage 1 address translation is disabled
SetInGuardedPage(FALSE);

// Output Address
FullAddress oa;
oa.address = va<55:0>;
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;

MemoryAttributes memattrs;
if regime == Regime\_EL10 && EL2Enabled() && walkparams.dc == '1' then
    MemAttrHints default_cacheability;
    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 walkparams.mtx == '1' then
        memattrs.tags = MemTag\_CanonicallyTagged;
    else
        memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '0';
elsif accdesc.acctype == AccessType\_IFETCH then
    MemAttrHints i_cache_attr;
    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;

    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);
elsif accdesc.acctype == AccessType\_TRBE && EffectiveTRBLIMITR\_EL1\_nVM() == '1' then
    memattrs = S1DecodeMemAttrs(TRBMAR_EL1.Attr, TRBMAR_EL1.SH, TRUE, walkparams);
else
    memattrs.memtype = MemType\_Device;
    memattrs.device = DeviceType\_nGnRnE;
    memattrs.shareability = Shareability\_OSH;
    if walkparams.mtx == '1' then
        memattrs.tags = MemTag\_CanonicallyTagged;
    else
        memattrs.tags = MemTag\_Untagged;
    memattrs.xs = '1';

```



```

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(va<55>, 4) else '0000';

fault.level = 0;
constant AddressSize addrtop = AArch64.AddrTop(walkparams.tbid, accdesc.acctype,
                                             walkparams.tbi);
constant AddressSize pamax = AArch64.PAMax();

if !IsZero(va<addrtop:pamax>) then
    fault.statuscode = Fault\_AddressSize;
elseif AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmd,
                                                memattrs) then
    fault.statuscode = Fault\_Alignment;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);
else
    ipa = CreateAddressDescriptor(va_in, oa, memattrs);
    ipa.mecid = AArch64.S1DisabledOutputMECID(walkparams, regime, ipa.paddress.paspace);
    return (fault, ipa);

```



```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                    bits(64) va, boolean aligned,
                                                    AccessDescriptor accdesc)

FaultRecord fault = 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);

walkparams = AArch64.GetS1TTWParams(regime, accdesc.ss, va);

constant integer slmintxsz = AArch64.S1MinTxSZ(regime, walkparams.dl28,
                                             walkparams.ds, walkparams.tgx);
constant integer slmaxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
if AArch64.S1TxSZFaults(regime, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);
elseif UInt(walkparams.txsz) < slmintxsz then
    walkparams.txsz = slmintxsz<5:0>;
elseif UInt(walkparams.txsz) > slmaxtxsz then
    walkparams.txsz = slmaxtxsz<5:0>;

if AArch64.VAIsOutOfRange(va, accdesc.acctype, regime, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if accdesc.el == ELO && walkparams.e0pd == '1' then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if (IsFeatureImplemented(FEAT_TME) && accdesc.el == ELO && walkparams.nfd == '1' &&
    accdesc.transactional) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

if (IsFeatureImplemented(FEAT_SVE) && accdesc.el == ELO && walkparams.nfd == '1' &&
    ((accdesc.nonfault && accdesc.contiguous) ||
     (accdesc.firstfault && !accdesc.first && !accdesc.contiguous))) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AccessDescriptor UNKNOWN);

AccessDescriptor descipaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.dl28 == '1' then
        (fault, descipaddr, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va,
                                                                    regime, accdesc, 128);
    else
        (fault, descipaddr, walkstate, descriptor<63:0>) = AArch64.S1Walk(fault, walkparams,
                                                                    va, regime, accdesc,
                                                                    64);

        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault\_None then
        return (fault, AccessDescriptor UNKNOWN);

if accdesc.acctype == AccessType\_IFETCH then

```

```

    // Flag the fetched instruction is from a guarded page
    SetInGuardedPage(walkstate.guardedpage == '1');

    if AArch64.S1HasAlignmentFaultDueToMemType(regime, accdesc, aligned, walkparams.ntlsmid,
        walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode == Fault\_None then
        fault = AArch64.S1CheckPermissions(fault, regime, walkstate, walkparams, accdesc);

    new_desc = descriptor;
    if AArch64.SetAccessFlag(walkparams.ha, accdesc, fault) then
        // Set descriptor AF bit
        new_desc<10> = '1';

    // 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.SetDirtyFlag(walkparams.hd, (walkparams.pie OR descriptor<51>),
        accdesc, fault) then
        // Clear descriptor AP[2]/nDirty bit permitting stage 1 writes
        new_desc<7> = '0';

    // Either the access flag was clear or AP[2]/nDirty is set
    if new_desc != descriptor then
        AddressDescriptor descpaddr;
        descaccess = CreateAccDescTTEUpdate(accdesc);
        if regime == Regime\_EL10 && EL2Enabled() then
            FaultRecord s2fault;
            slaarch64 = TRUE;
            s2aligned = TRUE;
            (s2fault, descpaddr) = AArch64.S2Translate(fault, descipaddr, slaarch64, s2aligned,
                descaccess);

            if s2fault.statuscode != Fault\_None then
                return (s2fault, AddressDescriptor UNKNOWN);

        else
            descpaddr = descipaddr;
            if walkparams.d128 == '1' then
                (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                    walkparams.tee, descaccess, descpaddr);
            else
                (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
                    new_desc<63:0>, walkparams.tee,
                    descaccess, descpaddr);

            mem_desc<127:64> = Zeros(64);

            if fault.statuscode != Fault\_None then
                if (accdesc.acctype == AccessType\_AT &&
                    !(boolean IMPLEMENTATION_DEFINED "AT reports the HW update fault")) then
                    // Mask the fault
                    fault.statuscode = Fault\_None;
                else
                    return (fault, AddressDescriptor UNKNOWN);

    until new_desc == descriptor || mem_desc == new_desc;

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);

    // Output Address
    oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);
    MemoryAttributes memattrs;
    if (accdesc.acctype == AccessType\_IFETCH &&
        (walkstate.memattrs.memtype == MemType\_Device || !AArch64.S1ICacheEnabled(regime))) then
        // Treat memory attributes as Normal Non-Cacheable
        memattrs = NormalNCMemAttr();
    memattrs.xs = walkstate.memattrs.xs;

```

```

elseif (accdesc.acctype != AccessType\_IFETCH && !AArch64.S1DCacheEnabled(regime) &&
        walkstate.memattrs.memtype == MemType\_Normal) 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.
    elseif (IsFeatureImplemented(FEAT_MTE_CANONICAL_TAGS) &&
        walkstate.memattrs.tags == MemTag\_CanonicallyTagged) then
        memattrs.tags = MemTag\_CanonicallyTagged;
else
    memattrs = walkstate.memattrs;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    memattrs.shareability = walkstate.memattrs.shareability;
else
    memattrs.shareability = EffectiveShareability(memattrs);

ipa = CreateAddressDescriptor(va, oa, memattrs);
ipa.slassured = walkstate.slassured;
varange = AArch64.GetVARange(va);
ipa.mecid = AArch64.S1OutputMECID(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);

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) &&
            (boolean IMPLEMENTATION_DEFINED
              "LD64B or ST64B faults to cacheable non-iWBoWB memory")) then
            fault.statuscode = Fault\_Exclusive;
            return (fault, ipa);
        elseif !(memattrs.inner.attrs == MemAttr\_NC && memattrs.outer.attrs == MemAttr\_NC) then
            fault.statuscode = Fault\_Exclusive;
            return (fault, ipa);

return (fault, ipa);

```



```

// AArch64.S2Translate()
// =====
// Translate stage 1 IPA to PA and combine memory attributes

(FaultRecord, AddressDescriptor) AArch64.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    boolean slaarch64, boolean aligned,
                                                    AccessDescriptor accdesc)

walkparams = AArch64.GetS2TTWParams(accdesc.ss, ipa.paddress.paspace, slaarch64);
FaultRecord fault = fault_in;
boolean s2fslmro;
// 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.sltagnodata = 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);

constant integer s2mintxsz = AArch64.S2MinTxSZ(walkparams.d128, walkparams.ds,
                                              walkparams.tgx, slaarch64);
constant integer s2maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
if AArch64.S2TxSZFaults(walkparams, slaarch64) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);
elseif UInt(walkparams.txsz) < s2mintxsz then
    walkparams.txsz = s2mintxsz<5:0>;
elseif UInt(walkparams.txsz) > s2maxtxsz then
    walkparams.txsz = s2maxtxsz<5:0>;

if (walkparams.d128 == '0' &&
    (AArch64.S2InvalidSL(walkparams) || AArch64.S2InconsistentSL(walkparams))) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

if AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams) then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddr;
TTWState walkstate;
bits(128) descriptor;
bits(128) new_desc;
bits(128) mem_desc;
repeat
    if walkparams.d128 == '1' then
        (fault, descaddr, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams,
                                                                    accdesc, 128);
    else
        (fault, descaddr, walkstate, descriptor<63:0>) = AArch64.S2Walk(fault, ipa,
                                                                    walkparams, accdesc,
                                                                    64);

        descriptor<127:64> = Zeros(64);
    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN);

    if AArch64.S2HasAlignmentFaultDueToMemType(accdesc, aligned, walkstate.memattrs) then
        fault.statuscode = Fault\_Alignment;

    if fault.statuscode == Fault\_None then
        (fault, s2fslmro) = AArch64.S2CheckPermissions(fault, walkstate, walkparams, ipa,
                                                         accdesc);

```

```

new_desc = descriptor;
if AArch64.SetAccessFlag(walkparams.ha, accdesc, fault) then
    // Set descriptor AF bit
    new_desc<10> = '1';

// 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.SetDirtyFlag(walkparams.hd, (walkparams.s2pie OR descriptor<51>),
    accdesc, fault) then
    // Set descriptor S2AP[1]/Dirty bit permitting stage 2 writes
    new_desc<7> = '1';

// Either the access flag was clear or S2AP[1]/Dirty is clear
if new_desc != descriptor then
    if walkparams.hdbss == '1' && descriptor<7> == '0' && new_desc<7> == '1' then
        fault = AppendToHDBSS(fault, ipa.paddress, accdesc, walkparams, walkstate.level);

// 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 (HDBSSPROD_EL2.FSC != '101000' &&
    (!fault.hdbssf || IsExternalAbort(fault.statuscode))) then
    constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if walkparams.dl28 == '1' then
        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
            walkparams.ee, descaccess,
            descaddr);
    else
        (fault, mem_desc<63:0>) = AArch64.MemSwapTableDesc(fault, descriptor<63:0>,
            new_desc<63:0>,
            walkparams.ee,
            descaccess, descaddr);

        mem_desc<127:64> = Zeros(64);

    if fault.statuscode != Fault\_None then
        if (accdesc.acctype == AccessType\_AT &&
            !(boolean IMPLEMENTATION_DEFINED "AT reports the HW update fault")) then
            // Mask the fault
            fault.statuscode = Fault\_None;
        else
            return (fault, AddressDescriptor UNKNOWN);

until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

ipa_64 = ZeroExtend(ipa.paddress.address, 64);
// Output Address
oa = StageOA(ipa_64, walkparams.dl28, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if ((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 && !S2DCacheEnabled())) then
    // Treat memory attributes as Normal Non-Cacheable
    s2_memattrs = NormalNCMemAttr();
    s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

s2aarch64 = TRUE;
MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombinesS1MemAttrs(ipa.memattrs, s2_memattrs, s2aarch64);

```



```

else
    memattrs = s2_memattrs;

pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
pa.s2fslmro = s2fslmro;
pa.mecid = AArch64.S2OutputMECID(walkparams, pa.paddress.paspace, descriptor);

if (accdesc.atomicop && !IsWBShareable(s2_memattrs) &&
    ConstrainUnpredictableBool(Unpredictable Atomic MMU IMPDEF FAULT)) then
    fault.statuscode = Fault\_Exclusive;
    return (fault, pa);

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) &&
            (boolean IMPLEMENTATION_DEFINED
              "LD64B or ST64B faults to cacheable non-iWBoWB memory")) then
            fault.statuscode = Fault\_Exclusive;
            return (fault, ipa);
        elseif !(s2_memattrs.inner.attrs == MemAttr\_NC && s2_memattrs.outer.attrs == MemAttr\_NC) then
            fault.statuscode = Fault\_Exclusive;
            return (fault, ipa);

return (fault, pa);

```

#### 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

boolean AArch64.SetAccessFlag(bit ha, AccessDescriptor accdesc, FaultRecord fault)
    if ha == '0' || !AArch64.SettingAccessFlagPermitted(fault) then
        return FALSE;
    elseif accdesc.acctype == AccessType\_AT then
        return boolean IMPLEMENTATION_DEFINED "AT updates AF";
    elseif accdesc.acctype IN {AccessType\_DC, AccessType\_IC} then
        return boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations";
    else
        // Set descriptor AF bit
        return TRUE;

```

#### Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.SetDirtyFlag

```

// AArch64.SetDirtyFlag()
// =====
// Determine whether the dirty flag could be set by HW given the fault status

boolean AArch64.SetDirtyFlag(bit hd, bit db_present, AccessDescriptor accdesc, FaultRecord fault)
    if hd == '0' || !AArch64.SettingDirtyStatePermitted(fault) then
        return FALSE;
    elseif accdesc.acctype IN {AccessType\_AT, AccessType\_IC, AccessType\_DC} then
        return FALSE;
    elseif !accdesc.write then
        return FALSE;
    else
        return db_present == '1';

```

## 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

boolean AArch64.SettingAccessFlagPermitted(FaultRecord fault)
    if fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode IN {Fault\_Alignment, Fault\_Permission} then
        return ConstrainUnpredictableBool(Unpredictable\_AFUPDATE);
    else
        return FALSE;
```

## 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

boolean AArch64.SettingDirtyStatePermitted(FaultRecord fault)
    if fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode == Fault\_Alignment then
        return ConstrainUnpredictableBool(Unpredictable\_DBUPDATE);
    else
        return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_translation/AArch64.TranslateAddress

```
// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccessDescriptor accdesc,
                                           boolean aligned, integer size)
    if (SPESampleInFlight && ! accdesc.acctype IN {AccessType\_IFETCH,
                                                    AccessType\_TRBE,
                                                    AccessType\_SPE}) then
        SPEStartCounter(SPECounterPosTranslationLatency);

    AddressDescriptor result = AArch64.FullTranslate(va, accdesc, aligned);

    if !IsFault(result) && accdesc.acctype != AccessType\_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if (IsFeatureImplemented(FEAT_RME) && !IsFault(result) &&
        (accdesc.acctype != AccessType\_DC ||
         boolean IMPLEMENTATION_DEFINED "GPC Fault on DC operations")) 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;

    if !IsFault(result) && accdesc.acctype == AccessType\_IFETCH then
        result.fault = AArch64.CheckDebug(va, accdesc, size);

    if (SPESampleInFlight && ! accdesc.acctype IN {AccessType\_IFETCH,
                                                    AccessType\_TRBE,
                                                    AccessType\_SPE}) then
        SPEStopCounter(SPECounterPosTranslationLatency);

    // Update virtual address for abort functions
    result.vaddress = ZeroExtend(va, 64);

    return result;
```

## 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

boolean AArch64.BlockDescSupported(bit d128, bit ds, TGx tgx, integer level)
    case tgx 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);
    return FALSE;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bit d128, bits(N) descriptor)
    bit nT;
    if !IsFeatureImplemented(FEAT_BBM) then
        return FALSE;
    nT = if d128 == '1' then descriptor<6> else descriptor<16>;
    bbm_level = AArch64.BlockBBMSupportLevel();
    nT_faults = (boolean IMPLEMENTATION_DEFINED
                  "BBM level 1 or 2 support nT bit causes Translation Fault");

    return bbm_level IN {1, 2} && nT == '1' && nT_faults;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, bit d128, integer level, bits(N) descriptor)
    if d128 == '1' then
        if (tgx == TGx\_64KB && level == 1) || (tgx == TGx\_4KB && level == 0) then
            return '0'; // RES0
        else
            return descriptor<111>;
    // When using TGx 64KB and FEAT_LPA is implemented,
    // the Contiguous bit is RES0 for Block descriptors at level 1

    if tgx == TGx\_64KB && level == 1 then
        return '0'; // RES0

    // 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

    if tgx == TGx\_16KB && level == 1 then
        return '0'; // RES0

    if tgx == TGx\_4KB && level == 0 then
        return '0'; // RES0

    return descriptor<52>;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(N) descriptor, bit d128, bit ds,
                                           TGx tgx, integer level)

    if descriptor<0> == '0' then
        return DescriptorType\_Invalid;
    elsif d128 == '1' then
        constant bits(2) skl = descriptor<110:109>;
        if tgx IN {TGx\_16KB, TGx\_64KB} && UInt(skl) == 3 then
            return DescriptorType\_Invalid;

            constant integer effective_level = 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;
        else
            if descriptor<1> == '1' then
                if level == FINAL\_LEVEL then
                    return DescriptorType\_Leaf;
                else
                    return DescriptorType\_Table;
            elsif descriptor<1> == '0' then
                if AArch64.BlockDescSupported(d128, ds, tgx, level) then
                    return DescriptorType\_Leaf;
                else
                    return DescriptorType\_Invalid;
            unreachable();
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(N) descriptor,
                                         Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;

    bits (4) pi_index;
    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>;
        permissions.ppi = Elem[walkparams.pir, UInt(pi_index), 4];
        permissions.upi = Elem[walkparams.pire0, UInt(pi_index), 4];
        permissions.ndirty = descriptor<7>;
    else
        if regime == Regime\_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
            permissions.ap<2:1> = descriptor<7>:'0';
            permissions.pxn = descriptor<54>;
        elseif 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>;
            permissions.dbm = descriptor<51>;
    if IsFeatureImplemented(FEAT_S1POE) then
        if walkparams.dl28 == '1' then
            permissions.po_index = descriptor<124:121>;
        else
            permissions.po_index = '0':descriptor<62:60>;
    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_tentry/AArch64.S1ApplyTablePerms

```
// AArch64.S1ApplyTablePerms()
// =====
// Apply hierarchical permissions encoded in stage 1 table descriptors

Permissions AArch64.S1ApplyTablePerms(Permissions permissions_in, bits(64) descriptor,
                                       Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    bits(2) ap_table;
    bit pxn_table;
    bit uxn_table;
    bit xn_table;
    if regime == Regime\_EL10 && EL2Enabled() && walkparams.nv1 == '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;

    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_ttentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(N) descriptor, S2TTWParams walkparams)
    Permissions permissions;
    bits(4) s2pi_index;
    if walkparams.s2pie == '1' then
        if walkparams.dl28 == '1' then
            s2pi_index = descriptor<118:115>;
        else
            s2pi_index = descriptor<54:53,51,6>;
        permissions.s2pi = Elem[walkparams.s2pir, UInt(s2pi_index), 4];
        permissions.s2dirty = descriptor<7>;
    else
        permissions.s2ap = descriptor<7:6>;
        if walkparams.dl28 == '1' then
            permissions.s2xn = descriptor<118>;
        else
            permissions.s2xn = descriptor<54>;

        if IsFeatureImplemented(FEAT_XNX) then
            if walkparams.dl28 == '1' then
                permissions.s2xnx = descriptor<117>;
            else
                permissions.s2xnx = descriptor<53>;
        else
            permissions.s2xnx = '0';

        permissions.dbm = descriptor<51>;
    if IsFeatureImplemented(FEAT_S2POE) then
        if walkparams.dl28 == '1' then
            permissions.s2po_index = descriptor<124:121>;
        else
            permissions.s2po_index = descriptor<62:59>;
    return permissions;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S1InitialTTWState

```
// AArch64.S1InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 1

TTWState AArch64.S1InitialTTWState(S1TTWParams walkparams, bits(64) va, Regime regime,
                                   SecurityState ss)

    TTWState walkstate;
    FullAddress tablebase;
    Permissions permissions;
    bits(128) ttbr;

    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;

    tablebase.address = AArch64.S1TTBaseAddress(walkparams, regime, ttbr);

    permissions.ap_table = '00';
    if HasUnprivileged(regime) then
        permissions.uxn_table = '0';
        permissions.pxn_table = '0';
    else
        permissions.xn_table = '0';

    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;
    else
        walkstate.slassured = FALSE;
    walkstate.disch = walkparams.disch;

    return walkstate;
```





```

// AArch64.S1NextWalkStateLeaf()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLeaf(TTWState currentstate, boolean s2fslmro, Regime regime,
                                     SecurityState ss, S1TWPParams walkparams, bits(N) descriptor)

    TTWState    nextstate;
    FullAddress baseaddress;
    baseaddress.address = AArch64.LeafBase(descriptor, walkparams.dl28,
                                         walkparams.ds,
                                         walkparams.tgx, currentstate.level);

    if currentstate.baseaddress.paspace == PAS\_Secure then
        // Determine PA space of the block from NS bit
        bit ns;
        ns = if walkparams.dl28 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS\_Secure else PAS\_NonSecure;
    elseif currentstate.baseaddress.paspace == PAS\_Root then
        // Determine PA space of the block from NSE and NS bits
        bit nse;
        bit ns;
        <nse,ns> = if walkparams.dl28 == '1' then descriptor<11,127> else descriptor<11,5>;
        constant bit nse2 = '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;

    elseif (currentstate.baseaddress.paspace == PAS\_Realm &&
           regime IN {Regime\_EL2, Regime\_EL20}) then
        // Realm EL2 and EL2&0 regimes have a stage 1 NS bit
        bit ns;
        ns = if walkparams.dl28 == '1' then descriptor<127> else descriptor<5>;
        baseaddress.paspace = if ns == '0' then PAS\_Realm else PAS\_NonSecure;
    elseif 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;

    nextstate.istable      = FALSE;
    nextstate.level       = currentstate.level;
    nextstate.baseaddress = baseaddress;

    bits(4) attrindx;
    if walkparams.aie == '1' then
        if walkparams.dl28 == '1' then
            attrindx = descriptor<5:2>;
        else
            attrindx = descriptor<59,4:2>;
    else
        attrindx = '0':descriptor<4:2>;

    bits(2) sh;
    if walkparams.dl28 == '1' then
        sh = descriptor<9:8>;
    elseif walkparams.ds == '1' then
        sh = walkparams.sh;
    else
        sh = descriptor<9:8>;
    attr = AArch64.MAIRAttr(UInt(attrindx), walkparams.mair2, walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs     = S1DecodeMemAttrs(attr, sh, slaarch64, walkparams);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions,
                                                         descriptor, regime, walkparams);

    bit protectedbit;
    if walkparams.dl28 == '1' then

```

```

    protectedbit = descriptor<114>;
else
    protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
if (currentstate.slassured && s2fslmro && protectedbit == '1') then
    nextstate.slassured = TRUE;
else
    nextstate.slassured = FALSE;

if walkparams.pnch == '1' || currentstate.disch == '1' then
    nextstate.contiguous = '0';
else
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, walkparams.d128,
                                                currentstate.level, descriptor);
if !HasUnprivileged(regime) then
    nextstate.nG = '0';
elseif 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>;

if walkparams.d128 == '1' then
    nextstate.guardedpage = descriptor<113>;
else
    nextstate.guardedpage = descriptor<50>;

return nextstate;

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S1NextWalkStateTable

```
// AArch64.S1NextWalkStateTable()
// =====
// Decode stage 1 table descriptor to transition to the next level

TTWState AArch64.S1NextWalkStateTable(TTWState currentstate, boolean s2fslmro, Regime regime,
S1TTWParams walkparams, bits(N) descriptor)

TTWState    nextstate;
FullAddress tablebase;
constant bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';

tablebase.address = AArch64.NextTableBase(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
    bit nstable;
    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;

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;
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;
bit protectedbit;
if walkparams.d128 == '1' then
    protectedbit = descriptor<114>;
else
    protectedbit = if walkparams.pnch == '1' then descriptor<52> else '0';
if (currentstate.slassured && s2fslmro && protectedbit == '1') then
    nextstate.slassured = TRUE;
else
    nextstate.slassured = FALSE;
nextstate.disch = if walkparams.d128 == '1' then descriptor<112> else '0';

return nextstate;
```



```

// AArch64.S1Walk()
// =====
// Traverse stage 1 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S1Walk(FaultRecord fault_in,
                                                                    S1TTWParams walkparams,
                                                                    bits(64) va, Regime regime,
                                                                    AccessDescriptor accdesc,
                                                                    integer N)

FaultRecord fault = fault_in;
boolean aligned;

if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, accdesc.ss);
constant integer startlevel = walkstate.level;

if startlevel > 3 then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

bits(N) descriptor;
AddressDescriptor walkaddress;
bits(2) skl = '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;

// 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' &&
    !(boolean IMPLEMENTATION_DEFINED "Apply effective shareability at stage 1")) then
    walkaddress.memattrs.shareability = walkstate.memattrs.shareability;
else
    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

boolean s2fslmro = FALSE;

DescriptorType descctype;
FullAddress descaddress = AArch64.S1SLTTEEntryAddress(walkstate.level, walkparams, va,
                                                         walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault\_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    walkaddress.slassured = walkstate.slassured;

    constant boolean toplevel = walkstate.level == startlevel;
    constant VARange varange = AArch64.GetVARange(va);
    constant AccessDescriptor walkaccess = CreateAccDescS1TTW(toplevel, varange, accdesc);
    FaultRecord s2fault;
    AddressDescriptor s2walkaddress;
    if regime == Regime\_EL10 && EL2Enabled() then
        constant boolean slaarch64 = TRUE;

```

```

aligned    = TRUE;
(s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, slaarch64, aligned,
                                              walkaccess);

if s2fault.statuscode != Fault\_None then
    return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

s2fslmro = s2walkaddress.s2fslmro;
(fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, walkaccess,
                                       fault, N);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess,
                                       fault, N);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
            bits(N) UNKNOWN);

bits(N) new_descriptor;
repeat
    new_descriptor = descriptor;
    descstype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                             walkparams.tgx, walkstate.level);

    case descstype of
        when DescriptorType\_Table
            walkstate = AArch64.S1NextWalkStateTable(walkstate, s2fslmro,
                                                    regime, walkparams, descriptor);
            skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
            descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.d128, skl,
                                                walkparams.tgx, walkparams.txs, va,
                                                walkstate.baseaddress);

            // Detect Address Size Fault by Descriptor Address
            if AArch64.S1OAOutOfRange(descaddress.address, walkparams) then
                fault.statuscode = Fault\_AddressSize;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);

            if walkparams.haft == '1' then
                new_descriptor<10> = '1';
            if (walkparams.d128 == '1' && skl != '00' &&
                AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
                fault.statuscode = Fault\_Translation;
                return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                        bits(N) UNKNOWN);

        when DescriptorType\_Leaf
            walkstate = AArch64.S1NextWalkStateLeaf(walkstate, s2fslmro,
                                                    regime, accdesc.ss, walkparams,
                                                    descriptor);

        when DescriptorType\_Invalid
            fault.statuscode = Fault\_Translation;
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                    bits(N) UNKNOWN);

    otherwise
        Unreachable();

if new_descriptor != descriptor then
    AddressDescriptor descaddr;
    constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
    if regime == Regime\_EL10 && EL2Enabled() then
        constant boolean slaarch64 = TRUE;
        aligned    = TRUE;
        (s2fault, descpaddr) = AArch64.S2Translate(fault, walkaddress,
                                                  slaarch64, aligned,
                                                  descaccess);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, AddressDescriptor UNKNOWN,
                    TTWState UNKNOWN, bits(N) UNKNOWN);

```

```

        else
            descaddr = walkaddress;

        (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                    walkparams.ee, descaccess,
                                                    descaddr);

        if fault.statuscode != Fault None then
            return (fault, AddressDescriptor UNKNOWN,
                    TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until desctype == DescriptorType\_Leaf;

    constant FullAddress oa = StageOA(va, walkparams.d128, walkparams.tgx, walkstate);

    if (walkstate.contiguous == '1' &&
        AArch64.ContiguousBitFaults(walkparams.d128, walkparams.txsz, walkparams.tgx,
                                     walkstate.level)) then
        fault.statuscode = Fault Translation;
    elseif (desctype == DescriptorType\_Leaf && walkstate.level < FINAL\_LEVEL &&
           AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
        fault.statuscode = Fault Translation;
    elseif AArch64.S1AMECFault(walkparams, walkstate.baseaddress.paspace, regime, descriptor) then
        fault.statuscode = Fault Translation;
    // Detect Address Size Fault by final output
    elseif AArch64.S1OAOOutOfRange(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} ||
             boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault AccessFlag;

    if fault.statuscode != Fault None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    return (fault, walkaddress, walkstate, descriptor);

```

### Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2InitialTTWState

```

// AArch64.S2InitialTTWState()
// =====
// Set properties of first access to translation tables in stage 2

TTWState AArch64.S2InitialTTWState(SecurityState ss, S2TTWParams walkparams)
    TTWState walkstate;
    FullAddress tablebase;
    bits(128) ttbr;

    ttbr = ZeroExtend(VTTBR_EL2, 128);
    case ss of
        when SS\_NonSecure tablebase.paspace = PAS\_NonSecure;
        when SS\_Realm tablebase.paspace = PAS\_Realm;
    tablebase.address = AArch64.S2TTBaseAddress(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;

```



## 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

TTWState AArch64.S2NextWalkStateLeaf(TTWState currentstate, SecurityState ss,
                                     S2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(N) descriptor)

TTWState    nextstate;
FullAddress baseaddress;

if ss == SS\_Secure then
    baseaddress.paspace = AArch64.SS2OutputPASpace(walkparams, ipa.paddress.paspace);
elsif ss == SS\_Realm then
    bit ns;
    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;
baseaddress.address = AArch64.LeafBase(descriptor, walkparams.d128, walkparams.ds,
                                     walkparams.tgx, currentstate.level);

nextstate.istable      = FALSE;
nextstate.level        = currentstate.level;
nextstate.baseaddress = baseaddress;
nextstate.permissions = AArch64.S2ApplyOutputPerms(descriptor, walkparams);

s2_attr = descriptor<5:2>;
s2_sh   = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
s2_fnxs = descriptor<11>;
if walkparams.fwb == '1' then
    nextstate.memattrs = AArch64.S2ApplyFWBMemAttrs(ipa.memattrs, walkparams, descriptor);
    if s2_attr<3:1> == '111' then
        nextstate.permissions.s2tag_na = '1';
    else
        nextstate.permissions.s2tag_na = '0';
else
    s2aarch64 = 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';
nextstate.contiguous = AArch64.ContiguousBit(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';

return nextstate;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.S2NextWalkStateTable

```
// AArch64.S2NextWalkStateTable()
// =====
// Decode stage 2 table descriptor to transition to the next level

TTWState AArch64.S2NextWalkStateTable(TTWState currentstate, S2TTWParams walkparams,
                                       bits(N) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;
    constant bits(2) skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';

    tablebase.address = AArch64.NextTableBase(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;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs    = currentstate.memattrs;

    return nextstate;
```



```

// AArch64.S2Walk()
// =====
// Traverse stage 2 translation tables obtaining the final descriptor
// as well as the address leading to that descriptor

(FaultRecord, AddressDescriptor, TTWState, bits(N)) AArch64.S2Walk(FaultRecord fault_in,
                                                                    AddressDescriptor ipa,
                                                                    S2TTWParams walkparams,
                                                                    AccessDescriptor accdesc,
                                                                    integer N)

FaultRecord fault = fault_in;
ipa_64 = ZeroExtend(ipa.paddress.address, 64);

TTWState walkstate;
if accdesc.ss == SS\_Secure then
    walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
else
    walkstate = AArch64.S2InitialTTWState(accdesc.ss, walkparams);
constant integer startlevel = walkstate.level;

if startlevel > 3 then
    fault.statuscode = Fault\_Translation;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

bits(N) descriptor;
constant AccessDescriptor walkaccess = CreateAccDescS2TTW(accdesc);
AddressDescriptor walkaddress;
bits(2) skl = '00';

walkaddress.vaddress = ipa.vaddress;
walkaddress.mecid = AArch64.S2TTWalkMECID(walkparams.emec, accdesc.ss);

if !S2DCacheEnabled() then
    walkaddress.memattr = NormalNCMemAttr();
    walkaddress.memattr.xs = walkstate.memattr.xs;
else
    walkaddress.memattr = walkstate.memattr;

walkaddress.memattr.shareability = EffectiveShareability(walkaddress.memattr);

DescriptorType desctype;

// Initial lookup might index into concatenated tables
FullAddress descaddress = AArch64.S2SLTTEntireAddress(walkparams, ipa.paddress.address,
                                                         walkstate.baseaddress);

// Detect Address Size Fault by Descriptor Address
if AArch64.S2OAOuOfRange(descaddress.address, walkparams) then
    fault.statuscode = Fault\_AddressSize;
    fault.level      = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

repeat
    fault.level = walkstate.level;
    walkaddress.paddress = descaddress;
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, walkaccess, fault, N);

    if fault.statuscode != Fault\_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    bits(N) new_descriptor;
    repeat
        new_descriptor = descriptor;
        desctype = AArch64.DecodeDescriptorType(descriptor, walkparams.d128, walkparams.ds,
                                                  walkparams.tgx, walkstate.level);

        case desctype of
            when DescriptorType\_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);

```

```

    skl = if walkparams.d128 == '1' then descriptor<110:109> else '00';
    descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.d128, skl,
                                         walkparams.tgx, walkparams.txs, ipa_64,
                                         walkstate.baseaddress);

    // Detect Address Size Fault by table descriptor
    if AArch64.S2OAOutOfRange(descaddress.address, walkparams) then
        fault.statuscode = Fault AddressSize;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    if walkparams.haft == '1' then
        new_descriptor<10> = '1';

    if (walkparams.d128 == '1' && skl != '00' &&
        AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
        fault.statuscode = Fault Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN,
                bits(N) UNKNOWN);

    when DescriptorType Leaf
        walkstate = AArch64.S2NextWalkStateLeaf(walkstate, accdesc.ss, walkparams, ipa,
                                                descriptor);

    when DescriptorType Invalid
        fault.statuscode = Fault Translation;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);

    otherwise
        Unreachable();

    if new_descriptor != descriptor then
        constant AccessDescriptor descaccess = CreateAccDescTTEUpdate(accdesc);
        (fault, descriptor) = AArch64.MemSwapTableDesc(fault, descriptor, new_descriptor,
                                                         walkparams.ee, descaccess,
                                                         walkaddress);

        if fault.statuscode != Fault None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(N) UNKNOWN);
        until new_descriptor == descriptor;
    until desctype == DescriptorType Leaf;

    constant FullAddress oa = 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;
    elsif (desctype == DescriptorType Leaf && walkstate.level < FINAL\_LEVEL &&
        AArch64.BlocknTFaults(walkparams.d128, descriptor)) then
        fault.statuscode = Fault Translation;
    // Detect Address Size Fault by final output
    elsif AArch64.S2OAOutOfRange(oa.address, walkparams) then
        fault.statuscode = Fault AddressSize;
    // Check descriptor AF bit
    elsif (descriptor<10> == '0' && walkparams.ha == '0' &&
        (!accdesc.acctype IN {AccessType DC, AccessType IC} ||
        boolean IMPLEMENTATION_DEFINED "Generate access flag fault on IC/DC operations")) then
        fault.statuscode = Fault AccessFlag;

    return (fault, walkaddress, walkstate, descriptor);

```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.SS2InitialTTWState

```
// AArch64.SS2InitialTTWState()
// =====
// Set properties of first access to translation tables in Secure stage 2

TTWState AArch64.SS2InitialTTWState(S2TTWParams walkparams, PASpace ipaspace)
    TTWState walkstate;
    FullAddress tablebase;
    bits(128) ttbr;

    if ipaspace == PAS_Secure then
        ttbr = ZeroExtend(VSTTBR_EL2, 128);
    else
        ttbr = ZeroExtend(VTTBR_EL2, 128);

    if ipaspace == PAS_Secure then
        if walkparams.sw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;
    else
        if walkparams.nsw == '0' then
            tablebase.paspace = PAS_Secure;
        else
            tablebase.paspace = PAS_NonSecure;

    tablebase.address = AArch64.S2TTBaseAddress(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;
```

## Library pseudocode for aarch64/translation/vmsa\_walk/AArch64.SS2OutputPASpace

```
// AArch64.SS2OutputPASpace()
// =====
// Assign PA Space to output of Secure stage 2 translation

PASpace AArch64.SS2OutputPASpace(S2TTWParams walkparams, PASpace ipaspace)
    if ipaspace == PAS_Secure then
        if walkparams.<sw,sa> == '00' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
    else
        if walkparams.<sw,sa,nsw,nsa> == '0000' then
            return PAS_Secure;
        else
            return PAS_NonSecure;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.BBMSupportLevel

```
// AArch64.BBMSupportLevel()
// =====
// Returns the level of FEAT_BBM supported

integer AArch64.BlockBBMSupportLevel()
    if !IsFeatureImplemented(FEAT_BBM) then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.GetS1TTWParams

```
// AArch64.GetS1TTWParams()
// =====
// Returns stage 1 translation table walk parameters from respective controlling
// System registers.

S1TTWParams AArch64.GetS1TTWParams(Regime regime, SecurityState ss, bits(64) va)
    S1TTWParams walkparams;

    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(ss, varange);
        when Regime\_EL10   walkparams = AArch64.S1TTWParamsEL10(varange);

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.GetS2TTWParams

```
// AArch64.GetS2TTWParams()
// =====
// Gather walk parameters for stage 2 translation

S2TTWParams AArch64.GetS2TTWParams(SecurityState ss, PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

    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();

    return walkparams;
```

## 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.

VARange AArch64.GetVARange(bits(64) va)
    if va<55> == '0' then
        return VARange\_LOWER;
    else
        return VARange\_UPPER;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.HaveS1TG

```
// AArch64.HaveS1TG()
// =====
// Determine whether the given translation granule is supported for stage 1

boolean AArch64.HaveS1TG(TGx tgx)
    case tgx of
        when TGx\_4KB    return IsFeatureImplemented(FEAT_TGran4K);
        when TGx\_16KB   return IsFeatureImplemented(FEAT_TGran16K);
        when TGx\_64KB   return IsFeatureImplemented(FEAT_TGran64K);
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.HaveS2TG

```
// AArch64.HaveS2TG()
// =====
// Determine whether the given translation granule is supported for stage 2

boolean AArch64.HaveS2TG(TGx tgx)
    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);
        else
            return AArch64.HaveS1TG(tgx);
```

## 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

integer AArch64.MaxTxSZ(TGx tgx)
    if IsFeatureImplemented(FEAT_TTST) then
        case tgx of
            when TGx_4KB    return 48;
            when TGx_16KB   return 48;
            when TGx_64KB   return 47;
        else
            return 39;
```



## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.NSS2TTWParams

```
// AArch64.NSS2TTWParams()
// =====
// Gather walk parameters specific for Non-secure stage 2 translation

S2TTWParams AArch64.NSS2TTWParams(boolean slaarch64)
    S2TTWParams walkparams;

    walkparams.vm = HCR_EL2.VM OR HCR_EL2.DC;
    walkparams.tgx = AArch64.S2DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.ps = VTCR_EL2.PS;
    walkparams.irgn = VTCR_EL2.IRGN0;
    walkparams.orgn = VTCR_EL2.ORGNO;
    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;

    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_HAFDBS) then VTCR_EL2.HA else '0';
    walkparams.hd = if walkparams.ha == '1' then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx\_4KB && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = (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';
    walkparams.s2pir = if IsFeatureImplemented(FEAT_S2PIE) then S2PIR_EL2 else Zeros(64);
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
        walkparams.assuredonly = VTCR_EL2.AssuredOnly;
    else
        walkparams.assuredonly = '0';
    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';
    if (IsFeatureImplemented(FEAT_HDBSS) && walkparams.hd == '1' &&
        (!HaveEL(EL3) || SCR_EL3.HDBSSEn == '1')) then
        walkparams.hdbss = VTCR_EL2.HDBSS;
    else
        walkparams.hdbss = '0';

    return walkparams;
```

## 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 processor

AddressSize AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.RLS2TTWParams

```
// AArch64.RLS2TTWParams()
// =====
// Gather walk parameters specific for Realm stage 2 translation

S2TTWParams AArch64.RLS2TTWParams(boolean slaarch64)
    // Realm stage 2 walk parameters are similar to Non-secure
    S2TTWParams walkparams = AArch64.NSS2TTWParams(slaarch64);
    walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
        IsSCTLR2EL2Enabled() then SCTLR2_EL2.EMEC else '0');
    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DCacheEnabled

```
// AArch64.S1DCacheEnabled()
// =====
// Determine cacheability of stage 1 data accesses

boolean AArch64.S1DCacheEnabled(Regime regime)
    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';
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DecodeTG0

```
// AArch64.S1DecodeTG0()
// =====
// Decode stage 1 granule size configuration bits TG0

TGx AArch64.S1DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00' tgx = TGx\_4KB;
        when '01' tgx = TGx\_64KB;
        when '10' tgx = TGx\_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00' tgx = TGx\_4KB;
            when '01' tgx = TGx\_64KB;
            when '10' tgx = TGx\_16KB;

    return tgx;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1DecodeTG1

```
// AArch64.S1DecodeTG1()
// =====
// Decode stage 1 granule size configuration bits TG1

TGx AArch64.S1DecodeTG1(bits(2) tgl_in)
    bits(2) tgl = tgl_in;
    TGx tgx;

    if tgl == '00' then
        tgl = bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size";

    case tgl of
        when '10'    tgx = TGx\_4KB;
        when '11'    tgx = TGx\_64KB;
        when '01'    tgx = TGx\_16KB;

    if !AArch64.HaveS1TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG1 encoded granule size" of
            when '10'    tgx = TGx\_4KB;
            when '11'    tgx = TGx\_64KB;
            when '01'    tgx = TGx\_16KB;

    return tgx;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1E0POEnabled

```
// AArch64.S1E0POEnabled()
// =====
// Determine whether stage 1 unprivileged permission overlay is enabled

boolean AArch64.S1E0POEnabled(Regime regime, bit nv1)
    assert HasUnprivileged(regime);

    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    case regime of
        when Regime\_EL20 return IsTCR2EL2Enabled() && TCR2_EL2.E0POE == '1';
        when Regime\_EL10 return IsTCR2EL1Enabled() && nv1 == '0' && TCR2_EL1.E0POE == '1';
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1EPD

```
// AArch64.S1EPD()
// =====
// Determine whether stage 1 translation table walk is allowed for the VA range

bit AArch64.S1EPD(Regime regime, bits(64) va)
    assert HasUnprivileged(regime);
    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;
```

### 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

boolean AArch64.S1Enabled(Regime regime, AccessType acctype)
    if acctype == AccessType\_TRBE && EffectiveTRBLIMITR\_EL1\_nVM() == '1' then
        return FALSE;
    if acctype == AccessType\_SPE && EffectivePMBLIMITR\_EL1\_nVM() == '1' then
        return FALSE;
    case regime of
        when Regime\_EL3 return SCTLR_EL3.M == '1';
        when Regime\_EL2 return SCTLR_EL2.M == '1';
        when Regime\_EL20 return SCTLR_EL2.M == '1';
        when Regime\_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLR_EL1.M == '1';
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1ICacheEnabled

```
// AArch64.S1ICacheEnabled()
// =====
// Determine cacheability of stage 1 instruction fetches

boolean AArch64.S1ICacheEnabled(Regime regime)
    case regime of
        when Regime\_EL3 return SCTLR_EL3.I == '1';
        when Regime\_EL2 return SCTLR_EL2.I == '1';
        when Regime\_EL20 return SCTLR_EL2.I == '1';
        when Regime\_EL10 return SCTLR_EL1.I == '1';
```

### 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

integer AArch64.S1MinTxSZ(Regime regime, bit d128, bit ds, TGx tgx)
    if IsFeatureImplemented(FEAT_LVA3) && d128 == '1' then
        if HasUnprivileged(regime) then
            return 9;
        else
            return 8;
    if (IsFeatureImplemented(FEAT_LVA) && tgx == TGx\_64KB) || ds == '1' then
        return 12;

    return 16;
```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1POEnabled

```
// AArch64.S1POEnabled()
// =====
// Determine whether stage 1 privileged permission overlay is enabled

boolean AArch64.S1POEnabled(Regime regime)
    if !IsFeatureImplemented(FEAT_S1POE) then
        return FALSE;

    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';
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1POR

```
// AArch64.S1POR()
// =====
// Identify stage 1 permissions overlay register for the acting translation regime

S1PORType AArch64.S1POR(Regime regime)
    case regime of
        when Regime\_EL3    return POR_EL3;
        when Regime\_EL2    return POR_EL2;
        when Regime\_EL20   return POR_EL2;
        when Regime\_EL10   return POR_EL1;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

bits(128) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime\_EL3    return ZeroExtend(TTBR0_EL3, 128);
        when Regime\_EL2    return ZeroExtend(TTBR0_EL2, 128);
        when Regime\_EL20
            if varange == VARange\_LOWER then
                return ZeroExtend(TTBR0_EL2, 128);
            else
                return ZeroExtend(TTBR1_EL2, 128);
        when Regime\_EL10
            if varange == VARange\_LOWER then
                return ZeroExtend(TTBR0_EL1, 128);
            else
                return ZeroExtend(TTBR1_EL1, 128);
```



```

// AArch64.S1TTWParamsEL10()
// =====
// Gather stage 1 translation table walk parameters for EL1&0 regime
// (with EL2 enabled or disabled)

S1TTWParams AArch64.S1TTWParamsEL10(VARange varange)
    S1TTWParams walkparams;

    if IsFeatureImplemented(FEAT_D128) && IsTCR2EL1Enabled() then
        walkparams.d128 = TCR2_EL1.D128;
    else
        walkparams.d128 = '0';
    constant bits(3) nvs = EffectiveHCR_EL2_NVx();
    walkparams.nv1 = nvs<1>;

    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL1;
    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');
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL1;
        if walkparams.nv1 != '1' then
            walkparams.pire0 = PIRE0_EL1;
    if varange == VARange_LOWER then
        walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL1.TG0);
        walkparams.txsz = TCR_EL1.T0SZ;
        walkparams.irgn = TCR_EL1.IRGN0;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH0;
        walkparams.tbi = TCR_EL1.TBI0;

        walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) || IsFeatureImplemented(FEAT_TME)
            then TCR_EL1.NFD0 else '0');
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID0 else '0';
        walkparams.e0pd = if IsFeatureImplemented(FEAT_E0PD) then TCR_EL1.E0PD0 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';
            if walkparams.pie == '1' then walkparams.hpd = '1';
            if (AArch64.S1POEnabled(Regime_EL10) ||
                AArch64.S1E0POEnabled(Regime_EL10, walkparams.nv1)) then walkparams.hpd = '1';
        walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL1.MTX0 else '0';
        walkparams.sk1 = if walkparams.d128 == '1' then TTBR0_EL1.SK1 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';
        if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled() then
            walkparams.fngna = TCR2_EL1.FNGNA0;
        else
            walkparams.fngna = '0';
    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) || IsFeatureImplemented(FEAT_TME)
            then TCR_EL1.NFD1 else '0');
        walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if IsFeatureImplemented(FEAT_E0PD) then TCR_EL1.E0PD1 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';
    if walkparams.pie == '1' then walkparams.hpd = '1';
    if (AArch64.S1POEnabled\(Regime\_EL10\) ||
        AArch64.S1E0POEnabled\(Regime\_EL10, walkparams.nv1\)) then walkparams.hpd = '1';
walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL1.MTX1 else '0';
walkparams.skl = if walkparams.d128 == '1' then TTBR1_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';
if IsFeatureImplemented(FEAT_THE) && IsTCR2EL1Enabled\(\) then
    walkparams.fngna = TCR2_EL1.FNGNA1;
else
    walkparams.fngna = '0';

walkparams.mair = MAIR_EL1;
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';

if EL2Enabled\(\) then
    walkparams.dc = HCR_EL2.DC;
    walkparams.dct = if IsFeatureImplemented(FEAT_MTE2) then HCR_EL2.DCT else '0';

if IsFeatureImplemented(FEAT_LSMAOC) then
    walkparams.ntlsmid = SCTLR_EL1.nTlSMID;
else
    walkparams.ntlsmid = '1';

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_EL1.CMOW else '0';
walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL1.HA else '0';
walkparams.hd = if walkparams.ha == '1' then TCR_EL1.HD else '0';
if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
    walkparams.ds = TCR_EL1.DS;
else
    walkparams.ds = '0';
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL1.EPAN else '1';
else
    walkparams.epan = '0';
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL1Enabled\(\) then
    walkparams.pnch = TCR2_EL1.PnCH;
else
    walkparams.pnch = '0';
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL1Enabled\(\) then
    walkparams.haft = TCR2_EL1.HAFT;
else
    walkparams.haft = '0';
walkparams.emec = (if IsFeatureImplemented(FEAT_MEC) &&
    IsSCTLR2EL2Enabled\(\) then SCTLR2_EL2.EMEC else '0');

return walkparams;

```



## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTWParamsEL2

```
// AArch64.S1TTWParamsEL2()
// =====
// Gather stage 1 translation table walk parameters for EL2 regime

S1TTWParams AArch64.S1TTWParamsEL2(SecurityState ss)
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL2.TG0);
    walkparams.txsz = TCR_EL2.T0SZ;
    walkparams.ps = TCR_EL2.PS;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI;
    walkparams.mair = MAIR_EL2;
    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) && IsTCR2EL2Enabled() then TCR2_EL2.PIE
                     else '0');
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL2;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL2;
    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';

    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';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled(Regime EL2) then walkparams.hpd = '1';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA else '0';
    walkparams.hd = if walkparams.ha == '1' then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL2.MTX else '0';
    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';
    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';

    return walkparams;
```



```

// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(SecurityState ss, VARange varange)
S1TTWParams walkparams;

if IsFeatureImplemented(FEAT_D128) && IsTCR2EL2Enabled() then
    walkparams.d128 = TCR2_EL2.D128;
else
    walkparams.d128 = '0';
if walkparams.d128 == '1' then
    walkparams.pie = '1';
else
    walkparams.pie = (if IsFeatureImplemented(FEAT_S1PIE) &&
IsTCR2EL2Enabled() then TCR2_EL2.PIE else '0');
if IsFeatureImplemented(FEAT_S1PIE) then
    walkparams.pir = PIR_EL2;
    walkparams.pire0 = PIRE0_EL2;
if IsFeatureImplemented(FEAT_AIE) then
    walkparams.mair2 = MAIR2_EL2;
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.TOSZ;
    walkparams.irgn = TCR_EL2.IRGN0;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH0;
    walkparams.tbi = TCR_EL2.TBI0;

    walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) ||
        IsFeatureImplemented(FEAT_TME) then TCR_EL2.NFD0 else '0');
    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID0 else '0';
    walkparams.e0pd = if IsFeatureImplemented(FEAT_E0PD) then TCR_EL2.E0PD0 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';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled\(Regime\_EL20\) || AArch64.S1E0POEnabled\(Regime\_EL20, '0'\) then
            walkparams.hpd = '1';
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL2.MTX0 else '0';
    walkparams.skl = 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';
else
    walkparams.tgx = AArch64.S1DecodeTG1(TCR_EL2.TG1);
    walkparams.txsz = TCR_EL2.T1SZ;
    walkparams.irgn = TCR_EL2.IRGN1;
    walkparams.orgn = TCR_EL2.ORGNO;
    walkparams.sh = TCR_EL2.SH1;
    walkparams.tbi = TCR_EL2.TBI1;

    walkparams.nfd = (if IsFeatureImplemented(FEAT_SVE) || IsFeatureImplemented(FEAT_TME)
        then TCR_EL2.NFD1 else '0');
    walkparams.tbid = if IsFeatureImplemented(FEAT_PAuth) then TCR_EL2.TBID1 else '0';
    walkparams.e0pd = if IsFeatureImplemented(FEAT_E0PD) then TCR_EL2.E0PD1 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';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled\(Regime\_EL20\) || AArch64.S1E0POEnabled\(Regime\_EL20, '0'\) then
            walkparams.hpd = '1';
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL2.MTX1 else '0';
    walkparams.skl = 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';

walkparams.mair = MAIR_EL2;
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';

if IsFeatureImplemented(FEAT_LSMAOC) then
    walkparams.ntlsmd = SCTLR_EL2.nTLSMD;
else
    walkparams.ntlsmd = '1';

walkparams.cmow = if IsFeatureImplemented(FEAT_CMOW) then SCTLR_EL2.CMOW else '0';
walkparams.ha   = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL2.HA else '0';
walkparams.hd   = if walkparams.ha == '1' then TCR_EL2.HD else '0';
if walkparams.tgx IN {TGx\_4KB, TGx\_16KB} && IsFeatureImplemented(FEAT_LPA2) then
    walkparams.ds = TCR_EL2.DS;
else
    walkparams.ds = '0';
if IsFeatureImplemented(FEAT_PAN3) then
    walkparams.epan = if walkparams.pie == '0' then SCTLR_EL2.EPAN else '1';
else
    walkparams.epan = '0';
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' && IsTCR2EL2Enabled\(\) then
    walkparams.pnch = TCR2_EL2.PnCH;
else
    walkparams.pnch = '0';
if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' && IsTCR2EL2Enabled\(\) then
    walkparams.haft = TCR2_EL2.HAFT;
else
    walkparams.haft = '0';
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.AMEC0 else TCR2_EL2.AMEC1;
else
    walkparams.amec = '0';

return walkparams;

```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S1TTWParamsEL3

```
// AArch64.S1TTWParamsEL3()
// =====
// Gather stage 1 translation table walk parameters for EL3 regime

S1TTWParams AArch64.S1TTWParamsEL3()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.S1DecodeTG0(TCR_EL3.TG0);
    walkparams.txsz = TCR_EL3.T0SZ;
    walkparams.ps = TCR_EL3.PS;
    walkparams.irgn = TCR_EL3.IRGN0;
    walkparams.orgn = TCR_EL3.ORGNO;
    walkparams.sh = TCR_EL3.SH0;
    walkparams.tbi = TCR_EL3.TBI;
    walkparams.mair = MAIR_EL3;
    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';
    if IsFeatureImplemented(FEAT_S1PIE) then
        walkparams.pir = PIR_EL3;
    if IsFeatureImplemented(FEAT_AIE) then
        walkparams.mair2 = MAIR2_EL3;
    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';
        if walkparams.pie == '1' then walkparams.hpd = '1';
        if AArch64.S1POEnabled(Regime_EL3) then walkparams.hpd = '1';
    walkparams.ha = if IsFeatureImplemented(FEAT_HAFDBS) then TCR_EL3.HA else '0';
    walkparams.hd = if walkparams.ha == '1' then TCR_EL3.HD else '0';
    if walkparams.tgx IN {TGx 4KB, TGx 16KB} && IsFeatureImplemented(FEAT_LPA2) then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';
    walkparams.mtx = if IsFeatureImplemented(FEAT_MTE4) then TCR_EL3.MTX else '0';
    if IsFeatureImplemented(FEAT_THE) && walkparams.d128 == '0' then
        walkparams.pnch = TCR_EL3.PnCH;
    else
        walkparams.pnch = '0';
    if IsFeatureImplemented(FEAT_HAFT) && walkparams.ha == '1' then
        walkparams.haft = TCR_EL3.HAFT;
    else
        walkparams.haft = '0';
    walkparams.emec = if IsFeatureImplemented(FEAT_MEC) then SCTLR2_EL3.EMEC else '0';

    return walkparams;
```

## Library pseudocode for aarch64/translation/vmsa\_walkparams/AArch64.S2DecodeTG0

```
// AArch64.S2DecodeTG0()
// =====
// Decode stage 2 granule size configuration bits TG0

TGx AArch64.S2DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    TGx tgx;

    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size";

    case tg0 of
        when '00'    tgx = TGx\_4KB;
        when '01'    tgx = TGx\_64KB;
        when '10'    tgx = TGx\_16KB;

    if !AArch64.HaveS2TG(tgx) then
        case bits(2) IMPLEMENTATION_DEFINED "TG0 encoded granule size" of
            when '00'    tgx = TGx\_4KB;
            when '01'    tgx = TGx\_64KB;
            when '10'    tgx = TGx\_16KB;

    return tgx;
```

## 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

integer AArch64.S2MinTxSZ(bit d128, bit ds, TGx tgx, boolean slaarch64)
    integer ips;

    if AArch64.PAMax() == 56 then
        if d128 == '1' then
            ips = 56;
        elsif tgx == TGx\_64KB || ds == '1' then
            ips = 52;
        else
            ips = 48;
    elsif AArch64.PAMax() == 52 then
        if tgx == TGx\_64KB || ds == '1' then
            ips = 52;
        else
            ips = 48;
    else
        ips = AArch64.PAMax();

    integer min_txsz = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```



```

// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation

S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

walkparams.d128 = if IsFeatureImplemented(FEAT_D128) then VTCR_EL2.D128 else '0';
if ipaspace == PAS_Secure then
    walkparams.tgx = AArch64.S2DecodeTGO(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.TOSZ;
    if walkparams.d128 == '1' then
        walkparams.skl = VSTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
elseif ipaspace == PAS_NonSecure then
    walkparams.tgx = AArch64.S2DecodeTGO(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.TOSZ;
    if walkparams.d128 == '1' then
        walkparams.skl = VTTBR_EL2.SKL;
    else
        walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && IsFeatureImplemented(FEAT_LPA2) then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
else
    Unreachable();

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.ORGNO;
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_HAFDBS) then VTCR_EL2.HA else '0';
walkparams.hd = if walkparams.ha == '1' then VTCR_EL2.HD else '0';
if walkparams.tgx IN {TGx_4KB, TGx_16KB} && IsFeatureImplemented(FEAT_LPA2) then
    walkparams.ds = VTCR_EL2.DS;
else
    walkparams.ds = '0';
walkparams.cmow = (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';
walkparams.s2pir = if IsFeatureImplemented(FEAT_S2PIE) then S2PIR_EL2 else Zeros(64);
if IsFeatureImplemented(FEAT_THE) && walkparams.d128 != '1' then
    walkparams.assuredonly = VTCR_EL2.AssuredOnly;
else
    walkparams.assuredonly = '0';
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';
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';

return walkparams;

```

### Library pseudocode for aarch64/translation/vmsa\_walkparams/S2DCacheEnabled

```

// S2DCacheEnabled()
// =====
// Returns TRUE if Stage 2 Data access cacheability is enabled

boolean S2DCacheEnabled()
    return HCR_EL2.CD == '0';

```

### Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```

// ClearStickyErrors()
// =====

ClearStickyErrors()
    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

    // 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;
    EDSCR.ERR = '0';           // Clear cumulative error flag

    return;

```

### Library pseudocode for shared/debug/DebugTarget/DebugTarget

```

// DebugTarget()
// =====
// Returns the debug exception target Exception level

bits(2) DebugTarget()
    ss = CurrentSecurityState();
    return DebugTargetFrom(ss);

```

## Library pseudocode for shared/debug/DebugTarget/DebugTargetFrom

```
// DebugTargetFrom()
// =====

bits(2) DebugTargetFrom(SecurityState from_state)
    boolean route_to_el2;
    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');
    else
        route_to_el2 = FALSE;

    bits(2) target;
    if route_to_el2 then
        target = EL2;
    elsif HaveEL(EL3) && !HaveAArch64() && from_state == SS\_Secure then
        target = EL3;
    else
        target = EL1;

    return target;
```

## 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.

boolean DoubleLockStatus()
    if !IsFeatureImplemented(FEAT_DoubleLock) then
        return FALSE;
    elsif ELUsingAArch32(EL1) then
        return DBGOSDLR.DLK == '1' && DBGPRCR.CORENPDRQ == '0' && !Halted();
    else
        return OSDLR_EL1.DLK == '1' && DBGPRCR_EL1.CORENPDRQ == '0' && !Halted();
```

## Library pseudocode for shared/debug/OSLockStatus/OSLockStatus

```
// OSLockStatus()
// =====
// Returns the state of the OS Lock.

boolean OSLockStatus()
    return (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK) == '1';
```

## Library pseudocode for shared/debug/SoftwareLockStatus/Component

```
// Component
// =====
// Component Types.

enumeration Component {
    Component_ETE,
    Component_TRBE,
    Component_RAS,
    Component_GIC,
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```

### Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// GetAccessComponent()
// =====
// Returns the accessed component.

Component GetAccessComponent();
```

### Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    constant Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    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;
```

### Library pseudocode for shared/debug/amu/IsG1ActivityMonitorImplemented

```
// IsG1ActivityMonitorImplemented()
// =====
// Returns TRUE if a G1 activity monitor is implemented for the counter
// and FALSE otherwise.

boolean IsG1ActivityMonitorImplemented(integer i);
```

### Library pseudocode for shared/debug/amu/IsG1ActivityMonitorOffsetImplemented

```
// IsG1ActivityMonitorOffsetImplemented()
// =====
// Returns TRUE if a G1 activity monitor offset is implemented for the counter,
// and FALSE otherwise.

boolean IsG1ActivityMonitorOffsetImplemented(integer i);
```

### Library pseudocode for shared/debug/authentication/AccessState

```
// AccessState()
// =====
// Returns the Security state of the access.

SecurityState AccessState();
```

## Library pseudocode for shared/debug/authentication/AllowExternalDebugAccess

```
// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed, FALSE otherwise.

boolean AllowExternalDebugAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalDebugAccess\(AccessState\(\)\);

// AllowExternalDebugAccess()
// =====
// Returns TRUE if an external debug interface access to the External debug registers
// is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalDebugAccess(SecurityState access_state)
    // 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 access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if access_state == SS\_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
        if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

    if HaveEL\(EL3\) then
        EDAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EDAD else MDCR_EL3.EDAD;
        return EDAD_bit == '0';
    else
        return NonSecureOnlyImplementation\(\);
```

## Library pseudocode for shared/debug/authentication/AllowExternalPMSSAccess

```
// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed, FALSE otherwise.

boolean AllowExternalPMSSAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMSSAccess\(AccessState\(\)\);

// AllowExternalPMSSAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU Snapshot
// registers is allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMSSAccess(SecurityState access_state)
    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.
    bits(2) epmssad = if HaveEL\(EL3\) then MDCR_EL3.EPMSSAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && epmssad IN {'01','10'} then
        (-, epmssad) = ConstrainUnpredictableBits\(Unpredictable\_RESEPMSSAD, 2\);
        // The value returned by ConstrainUnpredictableBits() must be a
        // non-reserved value
        assert epmssad IN {'00','11'};

    case epmssad of
        when '00'
            if IsFeatureImplemented(FEAT_RME) then
                return access_state == SS\_Root;
            else
                return access_state == SS\_Secure;
        when '01'
            assert IsFeatureImplemented(FEAT_RME);
            return access_state IN {SS\_Root, SS\_Realm};
        when '10'
            assert IsFeatureImplemented(FEAT_RME);
            return access_state IN {SS\_Root, SS\_Secure};
        when '11'
            return TRUE;
```

## Library pseudocode for shared/debug/authentication/AllowExternalPMUAccess

```
// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed, FALSE otherwise.

boolean AllowExternalPMUAccess()
    // The access may also be subject to OS Lock, power-down, etc.
    return AllowExternalPMUAccess\(AccessState\(\)\);

// AllowExternalPMUAccess()
// =====
// Returns TRUE if an external debug interface access to the PMU registers is
// allowed for the given Security state, FALSE otherwise.

boolean AllowExternalPMUAccess(SecurityState access_state)
    // The access may also be subject to OS Lock, power-down, etc.
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<EPMAD,EPMAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if IsFeatureImplemented(FEAT_Debugv8p4) then
        if access_state == SS\_Secure then return TRUE;
    else
        if !ExternalInvasiveDebugEnabled\(\) then return FALSE;
        if ExternalSecureInvasiveDebugEnabled\(\) then return TRUE;

    if HaveEL\(EL3\) then
        EPMAD_bit = if ELUsingAArch32\(EL3\) then SDCR.EPMAD else MDCR_EL3.EPMAD;
        return EPMAD_bit == '0';
    else
        return NonSecureOnlyImplementation\(\);
```

## Library pseudocode for shared/debug/authentication/AllowExternalTraceAccess

```
// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is allowed, FALSE otherwise.

boolean AllowExternalTraceAccess()
    if !IsFeatureImplemented(FEAT_TRBE) then
        return TRUE;
    else
        return AllowExternalTraceAccess(AccessState());

// AllowExternalTraceAccess()
// =====
// Returns TRUE if an external Trace access to the Trace registers is allowed for the
// given Security state, FALSE otherwise.

boolean AllowExternalTraceAccess(SecurityState access_state)
    // The access may also be subject to OS lock, power-down, etc.
    if !IsFeatureImplemented(FEAT_TRBE) then return TRUE;
    assert IsFeatureImplemented(FEAT_Debugv8p4);
    if IsFeatureImplemented(FEAT_RME) then
        case MDCR_EL3.<ETADE,ETAD> of
            when '00' return TRUE;
            when '01' return access_state IN {SS\_Root, SS\_Secure};
            when '10' return access_state IN {SS\_Root, SS\_Realm};
            when '11' return access_state == SS\_Root;

    if access_state == SS\_Secure then return TRUE;
    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();
```

## Library pseudocode for shared/debug/authentication/Debug

```
// Debug authentication signals
// =====

Signal DBGEN;
Signal NIDEN;
Signal SPIDEN;
Signal SPNIDEN;
Signal RLPIDEN;
Signal RTPIDEN;
```

## 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.

boolean ExternalInvasiveDebugEnabled()
    return DBGEN == Signal\_High;
```

## Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugAllowed

```
// ExternalNoninvasiveDebugAllowed()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugAllowed()
    return ExternalNoninvasiveDebugAllowed(PSTATE.EL);

// ExternalNoninvasiveDebugAllowed()
// =====

boolean ExternalNoninvasiveDebugAllowed(bits(2) el)
    if !ExternalNoninvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(el);

    if ((ELUsingAArch32(EL3) || ELUsingAArch32(EL1)) && el == EL0 &&
        ss == SS\_Secure && SDER.SUNIDEN == '1') then
        return TRUE;

    case ss of
        when SS\_NonSecure return TRUE;
        when SS\_Secure   return ExternalSecureNoninvasiveDebugEnabled();
        when SS\_Realm    return ExternalRealmNoninvasiveDebugEnabled();
        when SS\_Root     return ExternalRootNoninvasiveDebugEnabled();
```

## 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.

boolean ExternalNoninvasiveDebugEnabled()
    return (IsFeatureImplemented(FEAT_Debugv8p4) || ExternalInvasiveDebugEnabled() ||
        NIDEN == Signal\_High);
```

## 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.

boolean ExternalRealmInvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalInvasiveDebugEnabled() && RLPIDEN == Signal\_High;
```

## 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.

boolean ExternalRealmNoninvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalRealmInvasiveDebugEnabled();
```



## 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.

boolean ExternalRootInvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return (ExternalInvasiveDebugEnabled() &&
        (!IsFeatureImplemented(FEAT_SEL2) || ExternalSecureInvasiveDebugEnabled()) &&
        ExternalRealmInvasiveDebugEnabled() &&
        RTPIDEN == Signal_High);
```

## 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.

boolean ExternalRootNoninvasiveDebugEnabled()
    if !IsFeatureImplemented(FEAT_RME) then return FALSE;
    return ExternalRootInvasiveDebugEnabled();
```

## 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.

boolean ExternalSecureInvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == Signal_High;
```

## 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.

boolean ExternalSecureNoninvasiveDebugEnabled()
    if !HaveSecureState() then return FALSE;
    if !IsFeatureImplemented(FEAT_Debugv8p4) then
        return (ExternalNoninvasiveDebugEnabled() &&
            (SPIDEN == Signal_High || SPNIDEN == Signal_High));
    else
        return ExternalSecureInvasiveDebugEnabled();
```

## Library pseudocode for shared/debug/authentication/IsAccessNonSecure

```
// IsAccessNonSecure()
// =====
// Returns TRUE when an access is Non-Secure

boolean IsAccessNonSecure()
    return !IsAccessSecure();
```

## Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// IsAccessSecure()
// =====
// Returns TRUE when an access is Secure

boolean IsAccessSecure();
```

## Library pseudocode for shared/debug/authentication/IsCorePowered

```
// IsCorePowered()
// =====
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.

boolean IsCorePowered();
```

## Library pseudocode for shared/debug/breakpoint/BreakpointInfo

```
// BreakpointInfo
// =====
// Breakpoint related fields.

type BreakpointInfo is (
    BreakpointType bptype, // Type of breakpoint matched
    boolean match,        // breakpoint match
    boolean mismatch      // breakpoint mismatch
)
```

## Library pseudocode for shared/debug/breakpoint/BreakpointType

```
// BreakpointType
// =====

enumeration BreakpointType {
    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.

(Constraint, bits(2), bit, bit, bits(2)) CheckValidStateMatch(bits(2) ssc_in, bit ssce_in,
                                                             bit hmc_in, bits(2) pxc_in,
                                                             boolean isbreakpnt)

    if !IsFeatureImplemented(FEAT_RME) then assert ssce_in == '0';
    boolean reserved = FALSE;
    bits(2) ssc = ssc_in;
    bit ssce = ssce_in;
    bit hmc = hmc_in;
    bits(2) pxc = pxc_in;

    // Values that are not allocated in any architecture version
    case hmc:ssc:ssce: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;

    // Match 'Usr/Sys/Svc' valid only for AArch32 breakpoints
    if (!isbreakpnt || !HaveAArch32EL(EL1)) && hmc:pxc == '000' && ssc != '11' then
        reserved = TRUE;

    // Both EL3 and EL2 are not implemented
    if !HaveEL(EL3) && !HaveEL(EL2) && (hmc != '0' || ssc != '00') then
        reserved = TRUE;

    // EL3 is not implemented
    if !HaveEL(EL3) && ssc IN {'01','10'} && hmc:ssc:pxc != '10100' then
        reserved = TRUE;

    // EL3 using AArch64 only
    if (!HaveEL(EL3) || !HaveAArch64()) && hmc:ssc:pxc == '11000' then
        reserved = TRUE;

    // EL2 is not implemented
    if !HaveEL(EL2) && hmc:ssc:pxc == '11100' then
        reserved = TRUE;

    // Secure EL2 is not implemented
    if !IsFeatureImplemented(FEAT_SEL2) && (hmc:ssc:pxc) IN {'01100','10100','x11x1'} then
        reserved = TRUE;

    if reserved then
        // If parameters are set to a reserved type, behaves as either disabled or a defined type
        Constraint c;
        (c, <hmc:ssc:ssce:pxc>) = ConstrainUnpredictableBits(Unpredictable_RESBPWPCTRL, 6);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, ssc, ssce, hmc, pxc);
```

### Library pseudocode for shared/debug/breakpoint/ContextAwareBreakpointRange

```
// ContextAwareBreakpointRange()
// =====
// Returns two numbers indicating the index of the first and last context-aware breakpoint.

(integer, integer) ContextAwareBreakpointRange()
    constant integer b = NumBreakpointsImplemented\(\);
    constant integer c = NumContextAwareBreakpointsImplemented\(\);

    if b <= 16 then
        return (b - c, b - 1);
    elsif c <= 16 then
        return (16 - c, 15);
    else
        return (0, c - 1);
```

### Library pseudocode for shared/debug/breakpoint/IsContextAwareBreakpoint

```
// IsContextAwareBreakpoint()
// =====
// Returns TRUE if DBGBCR_EL1[n] is a context-aware breakpoint.

boolean IsContextAwareBreakpoint(integer n)
    (lower, upper) = ContextAwareBreakpointRange\(\);
    return n >= lower && n <= upper;
```

### Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented.

integer NumBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of breakpoints";
```

### Library pseudocode for shared/debug/breakpoint/NumContextAwareBreakpointsImplemented

```
// NumContextAwareBreakpointsImplemented()
// =====
// Returns the number of context-aware breakpoints implemented.

integer NumContextAwareBreakpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of context-aware breakpoints";
```

### Library pseudocode for shared/debug/breakpoint/NumWatchpointsImplemented

```
// NumWatchpointsImplemented()
// =====
// Returns the number of watchpoints implemented.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

### Library pseudocode for shared/debug/cti/CTI\_ProcessEvent

```
// CTI_ProcessEvent()
// =====
// Process a discrete event on a Cross Trigger output event trigger.

CTI_ProcessEvent(CrossTriggerOut id);
```

## Library pseudocode for shared/debug/cti/CTI\_SetEventLevel

```
// CTI_SetEventLevel()
// =====
// Set a Cross Trigger multi-cycle input event trigger to the specified level.

CTI_SetEventLevel(CrossTriggerIn id, Signal level);
```

## Library pseudocode for shared/debug/cti/CTI\_SignalEvent

```
// CTI_SignalEvent()
// =====
// Signal a discrete event on a Cross Trigger input event trigger.

CTI_SignalEvent(CrossTriggerIn id);
```

## Library pseudocode for shared/debug/cti/CrossTrigger

```
// CrossTrigger
// =====

enumeration CrossTriggerOut {CrossTriggerOut_DebugRequest, CrossTriggerOut_RestartRequest,
                             CrossTriggerOut_IRQ,           CrossTriggerOut_RSVD3,
                             CrossTriggerOut_TraceExtIn0,    CrossTriggerOut_TraceExtIn1,
                             CrossTriggerOut_TraceExtIn2,    CrossTriggerOut_TraceExtIn3};

enumeration CrossTriggerIn  {CrossTriggerIn_CrossHalt,      CrossTriggerIn_PMUOverflow,
                             CrossTriggerIn_RSVD2,          CrossTriggerIn_RSVD3,
                             CrossTriggerIn_TraceExtOut0,    CrossTriggerIn_TraceExtOut1,
                             CrossTriggerIn_TraceExtOut2,    CrossTriggerIn_TraceExtOut3};
```

## Library pseudocode for shared/debug/dccanditr/CheckForDCCInterrupts

```
// CheckForDCCInterrupts()
// =====

CheckForDCCInterrupts()
    commrx = (EDSCR.RXfull == '1');
    commtx = (EDSCR.TXfull == '0');

    // COMMRX and COMMTX support is optional and not recommended for new designs.
    // SetInterruptRequestLevel(InterruptID_COMMRX, if commrx then Signal_High else Signal_Low);
    // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then Signal_High else Signal_Low);

    // The value to be driven onto the common COMMIRQ signal.
    boolean commirq;
    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'));
    SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then Signal\_High else Signal\_Low);

    return;
```

## Library pseudocode for shared/debug/dccanditr/DTR

```
// DTR
// ===

bits(32) DTRRX;
bits(32) DTRTX;
```

## Library pseudocode for shared/debug/dccanditr/Read\_DBGDTRRX\_EL0

```
// Read_DBGDTRRX_EL0()
// =====
// Called on reads of debug register 0x080.

bits(32) Read_DBGDTRRX_EL0(boolean memory_mapped)
    return DTRRX;
```

## Library pseudocode for shared/debug/dccanditr/Read\_DBGDTRTX\_EL0

```
// Read_DBGDTRTX_EL0()
// =====
// Called on reads of debug register 0x08C.

bits(32) Read_DBGDTRTX_EL0(boolean memory_mapped)
    underrun = EDSCR.TXfull == '0' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0');
    value = if underrun then bits(32) UNKNOWN else DTRTX;

    if EDSCR.ERR == '1' then return value;                // Error flag set: no side-effects

    if underrun then
        EDSCR.TXU = '1'; EDSCR.ERR = '1';                // Underrun condition: block side-effects
        return value;                                     // Return UNKNOWN

    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"
    // If the load aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
    if EDSCR.ERR == '1' then
        EDSCR.TXfull = bit UNKNOWN;
        DBGDTRTX_EL0 = bits(64) UNKNOWN;
    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"
            // "MSR DBGDTRTX_EL0,X1" calls Write_DBGDTR_EL0() which sets TXfull.
            assert EDSCR.TXfull == '1';
        if !UsingAArch32() then
            X[1, 64] = bits(64) UNKNOWN;
        else
            R[1] = bits(32) UNKNOWN;
        EDSCR.ITE = '1';                                // See comments in Write_EDITR()

    return value;
```

## Library pseudocode for shared/debug/dccanditr/Read\_DBGDTR\_EL0

```
// Read_DBGDTR_EL0()
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) Read_DBGDTR_EL0(integer N)
    // 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};
    bits(N) result;
    if EDSCR.RXfull == '0' then
        result = bits(N) UNKNOWN;
    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;
        result<31:0> = DTRRX;
    EDSCR.RXfull = '0';
    return result;
```

## Library pseudocode for shared/debug/dccanditr/Write\_DBGDTRRX\_EL0

```
// Write_DBGDTRRX_EL0()
// =====
// Called on writes to debug register 0x080.

Write_DBGDTRRX_EL0(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return; // Error flag set: ignore write

    if EDSCR.RXfull == '1' || (Halted() && EDSCR.MA == '1' && EDSCR.ITE == '0') then
        EDSCR.RXO = '1'; EDSCR.ERR = '1'; // Overrun condition: ignore write
        return;

    EDSCR.RXfull = '1';
    DTRRX = value;

    if Halted() && EDSCR.MA == '1' then
        EDSCR.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[1, 64] = bits(64) UNKNOWN;
        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] = bits(32) UNKNOWN;
        // If the store aborts, the Data Abort exception is taken and EDSCR.ERR is set to 1
        if EDSCR.ERR == '1' then
            EDSCR.RXfull = bit UNKNOWN;
            DBGDTRRX_EL0 = bits(64) UNKNOWN;
        else
            // "MRS X1,DBGDTRRX_EL0" calls Read_DBGDTR_EL0() which clears RXfull.
            assert EDSCR.RXfull == '0';

            EDSCR.ITE = '1'; // See comments in Write_EDITR()
    return;
```

## Library pseudocode for shared/debug/dccanditr/Write\_DBGDTRTX\_EL0

```
// Write_DBGDTRTX_EL0()
// =====
// Called on writes to debug register 0x08C.

Write_DBGDTRTX_EL0(boolean memory_mapped, bits(32) value)
    DTRTX = value;
    return;
```

## Library pseudocode for shared/debug/dccanditr/Write\_DBGDTR\_EL0

```
// Write_DBGDTR_EL0()
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

Write_DBGDTR_EL0(bits(N) value_in)
    bits(N) value = 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 EDSCR.TXfull == '1' then
        value = bits(N) UNKNOWN;
    // On a 64-bit write, implement a half-duplex channel
    if N == 64 then DTRRX = value<63:32>;
    DTRTX = value<31:0>; // 32-bit or 64-bit write
    EDSCR.TXfull = '1';
    return;
```

## Library pseudocode for shared/debug/dccanditr/Write\_EDITR

```
// Write_EDITR()
// =====
// Called on writes to debug register 0x084.

Write_EDITR(boolean memory_mapped, bits(32) value)
    if EDSCR.ERR == '1' then return; // Error flag set: ignore write

    if !Halted() then return; // Non-debug state: ignore write

    if EDSCR.ITE == '0' || EDSCR.MA == '1' then
        EDSCR.ITO = '1'; EDSCR.ERR = '1'; // Overrun condition: block write
        return;

    // ITE indicates whether the processor is ready to accept another instruction; the processor
    // 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*/);

    EDSCR.ITE = '1';

    return;
```





```

// DCPSInstruction()
// =====
// Operation of the DCPS instruction in Debug state

DCPSInstruction(bits(2) target_el)

    SynchronizeContext();

bits(2) handle_el;
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;
    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;
    when EL3
        if EDSCR.SDD == '1' || !HaveEL(EL3) then
            UNDEFINED;
        else
            handle_el = EL3;
    otherwise
        Unreachable();

from_secure = CurrentSecurityState() == SS_Secure;
if ELUsingAArch32(handle_el) then
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    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';
        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';
    if handle_el == EL2 then
        ELR_hyp = bits(32) UNKNOWN; HSR = bits(32) UNKNOWN;
    else
        LR = bits(32) UNKNOWN;
        SPSR_curr[] = bits(32) UNKNOWN;
        PSTATE.E = SCTLR_ELx[].EE;
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

else // Targeting AArch64
    from_32 = UsingAArch32();
    if from_32 then AArch64.MaybeZeroRegisterUppers();
    if from_32 && IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' then
        ResetSVEState();
    else
        MaybeZeroSVEUppers(target_el);
    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';
    ELR_ELx[] = bits(64) UNKNOWN; SPSR_ELx[] = bits(64) UNKNOWN; ESR_ELx[] = bits(64) UNKNOWN;
    DLR_ELO = bits(64) UNKNOWN; DSPSR_ELO = bits(64) UNKNOWN;
    if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = '0';
    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
    if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = '0';
    if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLRL_ELx[].IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
        sync_errors = (sync_errors ||
            (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // The Effective value of SCTLRL[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();
    return;

```

### Library pseudocode for shared/debug/halting/DRPSInstruction

```

// DRPSInstruction()
// =====
// Operation of the A64 DRPS and T32 ERET instructions in Debug state

DRPSInstruction()

    sync_errors = IsFeatureImplemented(FEAT_IESB) && SCTLRL_ELx[].IESB == '1';
    if IsFeatureImplemented(FEAT_DoubleFault) && !UsingAArch32() then
        sync_errors = (sync_errors ||
            (EffectiveEA() == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3));
    // The Effective value of SCTLRL[].IESB might be zero in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    SynchronizeContext();

    DebugRestorePSR();

    return;

```

### Library pseudocode for shared/debug/halting/DebugHalt

```

// DebugHalt
// =====
// Reason codes for entry to Debug state

constant bits(6) DebugHalt_Breakpoint      = '000111';
constant bits(6) DebugHalt_EDBGRQ         = '010011';
constant bits(6) DebugHalt_Step_Normal     = '011011';
constant bits(6) DebugHalt_Step_Exclusive  = '011111';
constant bits(6) DebugHalt_OSUnlockCatch   = '100011';
constant bits(6) DebugHalt_ResetCatch      = '100111';
constant bits(6) DebugHalt_Watchpoint      = '101011';
constant bits(6) DebugHalt_HaltInstruction = '101111';
constant bits(6) DebugHalt_SoftwareAccess  = '110011';
constant bits(6) DebugHalt_ExceptionCatch  = '110111';
constant bits(6) DebugHalt_Step_NoSyndrome = '111011';

```

## Library pseudocode for shared/debug/halting/DebugRestorePSR

```
// DebugRestorePSR()
// =====

DebugRestorePSR()
// 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
    constant bits(32) spsr = SPSR_curr[];
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
    // In AArch32, all instructions are T32 and unconditional.
    PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
    DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
else
    constant bits(64) spsr = SPSR_ELx[];
    SetPSTATEFromPSR(spsr);
    PSTATE.<N,Z,C,V,SS,D,A,I,F> = bits(9) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
UpdateEDSCRFields(); // Update EDSCR PE state flags
```

## Library pseudocode for shared/debug/halting/DisableITRAndResumeInstructionPrefetch

```
// DisableITRAndResumeInstructionPrefetch()
// =====

DisableITRAndResumeInstructionPrefetch();
```

## Library pseudocode for shared/debug/halting/ExecuteA64

```
// ExecuteA64()
// =====
// Execute an A64 instruction in Debug state.

ExecuteA64(bits(32) instr);
```

## Library pseudocode for shared/debug/halting/ExecuteT32

```
// ExecuteT32()
// =====
// Execute a T32 instruction in Debug state.

ExecuteT32(bits(16) hw1, bits(16) hw2);
```

## Library pseudocode for shared/debug/halting/ExitDebugState

```
// ExitDebugState()
// =====

ExitDebugState()
    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';

    bits(64) new_pc;
    bits(64) spsr;

    if UsingAArch32() then
        new_pc = ZeroExtend(DLR, 64);
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            spsr = DSPSR2 : DSPSR;
        else
            spsr = ZeroExtend(DSPSR, 64);
    else
        new_pc = DLR_EL0;
        spsr = DSPSR_EL0;

    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

    constant boolean branch_conditional = FALSE;
    if UsingAArch32() then
        if ConstrainUnpredictableBool(Unpredictable\_RESTARTALIGNPC) then new_pc<0> = '0';
        // AArch32 branch
        BranchTo(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> = bits(32) UNKNOWN;
            new_pc<1:0> = bits(2) UNKNOWN;
        if IsFeatureImplemented(FEAT_BRBE) then
            BRBEDebugStateExit(new_pc);
        // A type of branch that is never predicted
        BranchTo(new_pc, BranchType\_DBGEXIT, branch_conditional);

    (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
    EDPRSR.HALTED = '0';
    UpdateEDSCRFields(); // Stop signalling PE state
    DisableITRAndResumeInstructionPrefetch();

    return;
```



```

// Halt()
// =====

Halt(bits(6) reason)
    constant boolean is_async = FALSE;
    constant FaultRecord fault = NoFault();
    Halt(reason, is_async, fault);

// Halt()
// =====

Halt(bits(6) reason, boolean is_async, FaultRecord fault)
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure\_DBG, FALSE);

    CTI\_SignalEvent(CrossTriggerIn\_CrossHalt); // Trigger other cores to halt

    constant bits(64) preferred_restart_address = ThisInstrAddr(64);
    bits(64) spsr = GetPSRFromPSTATE(DebugState, 64);

    if (IsFeatureImplemented(FEAT_BTI) && !is_async &&
        ! reason IN {DebugHalt\_Step\_Normal, DebugHalt\_Step\_Exclusive,
                     DebugHalt\_Step\_NoSyndrome, DebugHalt\_Breakpoint,
                     DebugHalt\_HaltInstruction} &&
        ConstrainUnpredictableBool(Unpredictable\_ZEROBTYP)) then
        spsr<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr<31:0>;
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            DSPSR2 = spsr<63:32>;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr;
    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';
    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
    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> = bits(14) UNKNOWN;
    else
        PSTATE.<SS,SSBS,D,A,I,F> = bits(6) UNKNOWN;

    if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = '1';
    if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = '00';
    if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = '0';

```

```

PSTATE.IL = '0';
if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
StopInstructionPrefetchAndEnableITR();
(EDSCR.STATUS, EDPRSR.HALTED) = (reason, '1');
UpdateEDSCRFields(); // Update EDSCR PE state flags.
if IsFeatureImplemented(FEAT_EDHSR) then
    UpdateEDHSR(reason, fault); // Update EDHSR fields.
if !is_async then EndOfInstruction();
return;

```

### 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.

```

```

boolean HaltOnBreakpointOrWatchpoint()
    return HaltingAllowed() && EDSCR.HDE == '1' && OSLSR_EL1.OSLK == '0';

```

### Library pseudocode for shared/debug/halting/Halted

```

// Halted()
// =====
boolean Halted()
    return ! EDSCR.STATUS IN {'000001', '000010'}; // Halted

```

### Library pseudocode for shared/debug/halting/HaltingAllowed

```

// HaltingAllowed()
// =====
// Returns TRUE if halting is currently allowed, FALSE if halting is prohibited.
boolean HaltingAllowed()
    if Halted() || DoubleLockStatus() then
        return FALSE;
    ss = CurrentSecurityState();
    case ss of
        when SS\_NonSecure return ExternalInvasiveDebugEnabled();
        when SS\_Secure return ExternalSecureInvasiveDebugEnabled();
        when SS\_Root return ExternalRootInvasiveDebugEnabled();
        when SS\_Realm return ExternalRealmInvasiveDebugEnabled();

```

### Library pseudocode for shared/debug/halting/Restarting

```

// Restarting()
// =====
boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting

```

### Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```

// StopInstructionPrefetchAndEnableITR()
// =====
StopInstructionPrefetchAndEnableITR();

```



## Library pseudocode for shared/debug/halting/UpdateDbgAuthStatus

```
// UpdateDbgAuthStatus()
// =====
// Provides information about the state of the
// IMPLEMENTATION DEFINED authentication interface for debug.

UpdateDbgAuthStatus()
    bits(2) nsid, nsnid;
    bits(2) sid, snid;
    bits(2) rlid, rtid;
    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.

    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.

    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.

    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.

    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.

    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.

    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;
```

## Library pseudocode for shared/debug/halting/UpdateEDHSR

```
// UpdateEDHSR()
// =====
// Update EDHSR watchpoint related fields.

UpdateEDHSR(bits(6) reason, FaultRecord fault)
    bits(64) syndrome = Zeros(64);
    if reason == DebugHalt\_Watchpoint then
        if IsFeatureImplemented(FEAT_GCS) && fault.accessdesc.acctype == AccessType\_GCS then
            syndrome<40> = '1'; // GCS
        syndrome<23:0> = WatchpointRelatedSyndrome(fault);
        if IsFeatureImplemented(FEAT_Debugv8p9) then
            if fault.write then syndrome<6> = '1'; // WnR
            if fault.accessdesc.acctype IN {AccessType\_DC, AccessType\_IC, AccessType\_AT} then
                syndrome<8> = '1'; // CM
            if IsFeatureImplemented(FEAT_NV2) && fault.accessdesc.acctype == AccessType\_NV2 then
                syndrome<13> = '1'; // VNCR
    else
        syndrome = bits(64) UNKNOWN;

    EDHSR = syndrome;
```

## Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()
    if !Halted() then
        EDSCR.EL = '00';
        if IsFeatureImplemented(FEAT_RME) then
            // SDD bit.
            EDSCR.SDD = if ExternalRootInvasiveDebugEnabled() then '0' else '1';
            EDSCR.<NSE,NS> = bits(2) UNKNOWN;
        else
            // SDD bit.
            EDSCR.SDD = if ExternalSecureInvasiveDebugEnabled() then '0' else '1';
            EDSCR.NS = bit UNKNOWN;

        EDSCR.RW = '1111';
    else
        EDSCR.EL = PSTATE.EL;
        // Error Pending.
        if EL2Enabled() && HCR_EL2.<AMO,TGE> == '10' && PSTATE.EL IN {EL0,EL1} then
            EDSCR.A = if IsVirtualSErrorPending() then '1' else '0';
        else
            EDSCR.A = if IsPhysicalSErrorPending() then '1' else '0';

        ss = 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';
            else
                EDSCR.NS = if ss == SS_Secure then '0' else '1';

        bits(4) RW;
        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';
        if !HaveEL(EL2) || (HaveEL(EL3) && SCR_curr[].NS == '0' && !IsSecureEL2Enabled()) then
            RW<2> = RW<1>;
        else
            RW<2> = if ELUsingAArch32(EL2) then '0' else '1';
        if !HaveEL(EL3) then
            RW<3> = RW<2>;
        else
            RW<3> = if ELUsingAArch32(EL3) then '0' else '1';

        // The least-significant bits of EDSCR.RW are UNKNOWN if any higher EL is using AArch32.
        if RW<3> == '0' then RW<2:0> = bits(3) UNKNOWN;
        elsif RW<2> == '0' then RW<1:0> = bits(2) UNKNOWN;
        elsif RW<1> == '0' then RW<0> = bit UNKNOWN;
        EDSCR.RW = RW;
    return;
```

## Library pseudocode for shared/debug/haltingevents/CheckExceptionCatch

```
// CheckExceptionCatch()
// =====
// Check whether an Exception Catch debug event is set on the current Exception level

CheckExceptionCatch(boolean exception_entry)
    // 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.
    ss = SecurityStateAtEL(PSTATE.EL);
    integer base;

    case ss of
        when SS\_Secure      base = 0;
        when SS\_NonSecure  base = 4;
        when SS\_Realm      base = 16;
        when SS\_Root       base = 0;
    if HaltingAllowed() then
        boolean halt;
        if IsFeatureImplemented(FEAT_Debugv8p2) then
            exception_exit = !exception_entry;
            increment = if ss == SS\_Realm then 4 else 8;
            ctrl = EDECCR<UInt(PSTATE.EL) + base + increment>:EDECCR<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);
            else
                halt = (EDECCR<UInt(PSTATE.EL) + base> == '1');

            if halt then
                if IsFeatureImplemented(FEAT_Debugv8p8) && exception_entry then
                    EDESR.EC = '1';
                else
                    Halt(DebugHalt\_ExceptionCatch);
```

## Library pseudocode for shared/debug/haltingevents/CheckHaltingStep

```
// CheckHaltingStep()
// =====
// Check whether EDESR.SS has been set by Halting Step

CheckHaltingStep(boolean is_async)
    step_enabled = EDECR.SS == '1' && HaltingAllowed();
    active_pending = step_enabled && EDESR.SS == '1';
    if active_pending then
        if HaltingStep\_DidNotStep() then
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_NoSyndrome, is_async, fault);
        elsif HaltingStep\_SteppedEX() then
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_Exclusive, is_async, fault);
        else
            constant FaultRecord fault = NoFault();
            Halt(DebugHalt\_Step\_Normal, is_async, fault);
    if step_enabled then ShouldAdvanceHS = TRUE;
    return;
```

### Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((IsFeatureImplemented(FEAT_DoPD) && CTIDEVCTL.OSUCE == '1') ||
        (!IsFeatureImplemented(FEAT_DoPD) && EDECR.OSUCE == '1')) then
        if !Halted() then EDESR.OSUC = '1';
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingExceptionCatch

```
// CheckPendingExceptionCatch()
// =====
// Check whether EDESR.EC has been set by an Exception Catch debug event.

CheckPendingExceptionCatch(boolean is_async)
    if IsFeatureImplemented(FEAT_Debugv8p8) && HaltingAllowed() && EDESR.EC == '1' then
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt\_ExceptionCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingOSUnlockCatch

```
// CheckPendingOSUnlockCatch()
// =====
// Check whether EDESR.OSUC has been set by an OS Unlock Catch debug event

CheckPendingOSUnlockCatch()
    if HaltingAllowed() && EDESR.OSUC == '1' then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt\_OSUnlockCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckPendingResetCatch

```
// CheckPendingResetCatch()
// =====
// Check whether EDESR.RC has been set by a Reset Catch debug event

CheckPendingResetCatch()
    if HaltingAllowed() && EDESR.RC == '1' then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt\_ResetCatch, is_async, fault);
```

### Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    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);
```

## Library pseudocode for shared/debug/haltingevents/CheckSoftwareAccessToDebugRegisters

```
// CheckSoftwareAccessToDebugRegisters()
// =====
// Check for access to Breakpoint and Watchpoint registers.

CheckSoftwareAccessToDebugRegisters()
    os_lock = (if ELUsingAArch32(EL1) then DBGOSLSR.OSLK else OSLSR_EL1.OSLK);
    if HaltingAllowed() && EDCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt\_SoftwareAccess);
```

## Library pseudocode for shared/debug/haltingevents/CheckTRBEHalt

```
// CheckTRBEHalt()
// =====

CheckTRBEHalt()
    if !IsFeatureImplemented(FEAT_Debugv8p9) || !IsFeatureImplemented(FEAT_TRBE_EXT) then
        return;

    if (HaltingAllowed() && TraceBufferEnabled() &&
        TRBSR_EL1.IRQ == '1' && EDCR.TRBE == '1') then
        Halt(DebugHalt\_EDBGRQ);
```

## Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        constant boolean is_async = TRUE;
        constant FaultRecord fault = NoFault();
        Halt(DebugHalt\_EDBGRQ, is_async, fault);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

## Library pseudocode for shared/debug/haltingevents/HSAdvance

```
// HSAdvance()
// =====
// Advance the Halting Step State Machine

HSAdvance()
    if !ShouldAdvanceHS then return;
    step_enabled = EDCR.SS == '1' && HaltingAllowed();
    active_not_pending = step_enabled && EDCR.SS == '0';
    if active_not_pending then EDCR.SS = '1'; // set as pending.
    ShouldAdvanceHS = FALSE;
    return;
```

## 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.

boolean HaltingStep_DidNotStep();
```

## 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.

boolean HaltingStep_SteppedEX();
```

## Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    constant SecurityState ss = 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();
            else
                int_dis = FALSE;
        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());
            return int_dis;
```

## Library pseudocode for shared/debug/pmu

```
array integer PMUEventAccumulator[0..30]; // Accumulates PMU events for a cycle

array boolean PMULastThresholdValue[0..30]; // A record of the threshold result for each
```

## Library pseudocode for shared/debug/pmu/CYCLE\_COUNTER\_ID

```
// Constant used in PMU functions to represent actions on the cycle counter.
constant integer CYCLE_COUNTER_ID = 31;
```

## 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.

CheckForPMUOverflow()
    constant boolean include_r1 = TRUE;
    constant boolean include_r2 = TRUE;
    constant boolean include_r3 = TRUE;

    constant boolean enabled = PMUInterruptEnabled\(\);
    constant boolean pmuirq = CheckPMUOverflowCondition(PMUOverflowCondition\_IRQ,
                                                    include_r1, include_r2, include_r3);

    SetInterruptRequestLevel(InterruptID\_PMUIRQ,
                            if enabled && pmuirq then Signal\_High else Signal\_Low);
    CTI\_SetEventLevel(CrossTriggerIn\_PMUOverflow,
                    if pmuirq then Signal\_High else Signal\_Low);

    // 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.

    if IsFeatureImplemented(FEAT_PMUv3p9) && IsFeatureImplemented(FEAT_Debugv8p9) then
        if pmuirq && EDECR.PME == '1' then ExternalDebugRequest();

    if ShouldBRBEFreeze() then
        BRBEFreeze();

    return;
```





```

// 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)].

boolean CheckPMUOverflowCondition(PMUOverflowCondition reason,
                                boolean include_r1, boolean include_r2, boolean include_r3)

    // '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.
    // If 'exclude_sync' is TRUE, then counters in synchronous mode are NOT checked.
    boolean check_e;
    boolean check_inten;
    boolean exclude_cyc;
    boolean exclude_sync;

    case reason of
        when PMUOverflowCondition\_PMUException
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        when PMUOverflowCondition\_BRBEFreeze
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = TRUE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        when PMUOverflowCondition\_Freeze
            check_e      = FALSE;
            check_inten  = FALSE;
            exclude_cyc  = FALSE;
            exclude_sync = IsFeatureImplemented(FEAT_SEBEP);
        when PMUOverflowCondition\_IRQ
            check_e      = TRUE;
            check_inten  = TRUE;
            exclude_cyc  = FALSE;
            exclude_sync = FALSE;
        otherwise
            Unreachable();

    bits(64) ovsf;

    if HaveAArch64() then
        ovsf = PMOVSCLR_EL0;
        ovsf<63:33> = Zeros(31);
        if !IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
            ovsf<INSTRUCTION COUNTER ID> = '0';
    else
        ovsf = ZeroExtend(PMOVSR, 64);

    constant integer counters = NUM_PMU_COUNTERS;
    // Remove unimplemented counters - these fields are RES0
    if counters < 31 then
        ovsf<30:counters> = Zeros(31-counters);

    for idx = 0 to counters - 1
        bit global_en;
        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';
            when PMUCounterRange\_R2
                global_en = if HaveAArch64() then MDCR_EL2.HPME else HDCR.HPME;
                if !include_r2 then

```

```

        ovssf<idx> = '0';
    when PMUCounterRange\_R3
        global_en = PMCCR.EPME;
        if !include_r3 then
            ovssf<idx> = '0';
        otherwise
            Unreachable();
    if exclude_sync then
        constant bit sync = PMEVTYPERR_EL0[idx].SYNC;
        ovssf<idx> = ovssf<idx> AND NOT sync;
    if check_e then
        ovssf<idx> = ovssf<idx> AND global_en;

// Cycle counter
if exclude_cyc || !include_r1 then
    ovssf<CYCLE\_COUNTER\_ID> = '0';

if check_e then
    ovssf<CYCLE\_COUNTER\_ID> = ovssf<CYCLE\_COUNTER\_ID> AND PMCR_EL0.E;

// Instruction counter
if HaveAArch64() && IsFeatureImplemented(FEAT_PMUV3_ICNTR) then
    if !include_r1 then
        ovssf<INSTRUCTION\_COUNTER\_ID> = '0';
    if exclude_sync then
        constant bit sync = PMICFILTR_EL0.SYNC;
        ovssf<INSTRUCTION\_COUNTER\_ID> = ovssf<INSTRUCTION\_COUNTER\_ID> AND NOT sync;
    if check_e then
        ovssf<INSTRUCTION\_COUNTER\_ID> = ovssf<INSTRUCTION\_COUNTER\_ID> AND PMCR_EL0.E;

if check_inten then
    constant bits(64) inten = (if HaveAArch64() then PMINTENCLR_EL1
                                else ZeroExtend(PMINTENCLR, 64));
    ovssf = ovssf AND inten;

return !IsZero(ovssf);

```

## 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.

ClearEventCounters()
    // 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));

```



```

// 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).

boolean CountPMUEvents(integer idx)
    constant integer counters = NUM_PMU_COUNTERS;
    assert (idx == CYCLE\_COUNTER\_ID || idx < counters ||
            (idx == INSTRUCTION\_COUNTER\_ID && IsFeatureImplemented(FEAT_PMuV3_ICNTR)));

    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;

    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters
    constant PMUCounterRange counter_range = GetPMUCounterRange(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    bit global_en;
    bit counter_en;
    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();

    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>;
            // Event counter <n> does not count when all of the following are true:
            // - FEAT_SEBEP is implemented
            // - PMEVTYPER<n>_EL0.SYNC == 1
            // - Event counter <n> is configured to count an event that is not a synchronous event
            if (IsFeatureImplemented(FEAT_SEBEP) && PMEVTYPER_EL0[idx].SYNC == '1' &&
                !IsSupportingPMUSynchronousMode(PMEVTYPER_EL0[idx].evtCount)) then
                counter_en = '0';

    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';

    if prohibited && PSTATE.EL == EL0 then
        if ELUsingAArch32(EL3) then
            prohibited = SDER.SUNIDEN == '0';
        elseif ELUsingAArch32(EL1) then
            prohibited = SDER32_EL3.SUNIDEN == '0';

// 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));

// 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
    hpmd = if HaveAArch64() then MDCR_EL2.HPMD else HDCR.HPMD;
    prohibited = hpmd == '1';

// The IMPLEMENTATION DEFINED authentication interface might override software
if prohibited && !IsFeatureImplemented(FEAT_Debugv8p2) then
    prohibited = !ExternalSecureNoninvasiveDebugEnabled();

// If FEAT_PMUv3p7 is implemented, event counting can be frozen
if IsFeatureImplemented(FEAT_PMUv3p7) then
    bit fz;
    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();
    frozen = (fz == '1') && ShouldPMUFreeze(counter_range);
    frozen = frozen || SPEFreezeOnEvent(idx);
else
    frozen = FALSE;

// 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
    dp = 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;

// 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
    sccd = if HaveAArch64() then MDCR_EL3.SCCD else SDCR.SCCD;
    if sccd == '1' then
        prohibited = TRUE;

if PSTATE.EL == EL2 then
    hccd = if HaveAArch64() then MDCR_EL2.HCCD else HDCR.HCCD;
    if hccd == '1' then
        prohibited = TRUE;

// 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;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH, RLK, RLU, RLH} bits
bits(32) filter;
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];

p = filter<31>;
u = filter<30>;
nsk = if HaveEL(EL3) then filter<29> else '0';
nsu = if HaveEL(EL3) then filter<28> else '0';
nsh = if HaveEL(EL2) then filter<27> else '0';
m = if HaveEL(EL3) && HaveAArch64() then filter<26> else '0';
sh = if HaveEL(EL3) && IsFeatureImplemented(FEAT_SEL2) then filter<24> else '0';
rlk = if IsFeatureImplemented(FEAT_RME) then filter<22> else '0';
rlu = if IsFeatureImplemented(FEAT_RME) then filter<21> else '0';
rlh = 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;
    when EL1
        case ss of
            when SS_NonSecure filtered = p != nsk;
            when SS_Secure filtered = p == '1';
            when SS_Realm filtered = p != rlk;
    when EL2
        case ss of
            when SS_NonSecure filtered = nsh == '0';
            when SS_Secure filtered = nsh == sh;
            when SS_Realm filtered = nsh == rlh;
    when EL3
        if HaveAArch64() then
            filtered = m != p;
        else
            filtered = p == '1';

if IsFeatureImplemented(FEAT_PMuV3_SME) then
    constant boolean is_streaming_mode = PSTATE.SM == '1';
    bits(2) vs;
    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;

```

```

boolean streaming_mode_filtered;
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'));

filtered = filtered || streaming_mode_filtered;

return !debug && enabled && !prohibited && !filtered && !frozen;

```

### Library pseudocode for shared/debug/pmu/EffectiveEPMN

```

// EffectiveEPMN()
// =====
// Returns the Effective value of PMCCR.EPMN.

bits(5) EffectiveEPMN()
    constant integer counters = NUM_PMU_COUNTERS;
    bits(5) epmn_bits;

    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        epmn_bits = PMCCR.EPMN;
        if UInt(epmn_bits) > counters then
            (-, epmn_bits) = ConstrainUnpredictableBits(Unpredictable\_RES\_EPMN, 5);
        else
            epmn_bits = counters<4:0>;
    else
        epmn_bits = counters<4:0>;

    return epmn_bits;

```

### Library pseudocode for shared/debug/pmu/EffectiveHPMN

```

// EffectiveHPMN()
// =====
// Returns the Effective value of MDCR_EL2.HPMN or HDCR.HPMN.

bits(5) EffectiveHPMN()
    constant integer counters = UInt(EffectiveEPMN());
    bits(5) hpmn_bits;

    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>;

    if (UInt(hpmn_bits) > counters ||
        (!IsFeatureImplemented(FEAT_HPMN0) && IsZero(hpmn_bits))) then
        (-, hpmn_bits) = ConstrainUnpredictableBits(Unpredictable\_RES\_HPMN, 5);
    else
        hpmn_bits = counters<4:0>;

    return hpmn_bits;

```



## Library pseudocode for shared/debug/pmu/GetNumEventCountersAccessible

```
// GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer GetNumEventCountersAccessible()
    integer n;

    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(EffectiveHPMN());
    else
        n = UInt(EffectiveEPMN());

    return n;
```

## Library pseudocode for shared/debug/pmu/GetNumEventCountersSelfHosted

```
// GetNumEventCountersSelfHosted()
// =====
// Return the number of event counters that can be accessed by the Self-hosted software.

integer GetNumEventCountersSelfHosted()
    if IsFeatureImplemented(FEAT_PMUv3_EXTPMN) then
        return UInt(EffectiveEPMN());
    else
        return NUM_PMU_COUNTERS;
```

## Library pseudocode for shared/debug/pmu/GetPMUAccessMask

```
// GetPMUAccessMask()
// =====
// Return a mask of the PMU counters accessible at the current Exception level

bits(64) GetPMUAccessMask()
    bits(64) mask = Zeros(64);

    // 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';

    // PMCCNTR_EL0 is always implemented and accessible
    mask<CYCLE_COUNTER_ID> = '1';

    // PMEVCNTR<n>_EL0
    constant integer counters = GetNumEventCountersAccessible();
    if counters > 0 then
        mask<counters-1:0> = Ones(counters);

    // 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

    return mask;
```

## Library pseudocode for shared/debug/pmu/GetPMUCounterRange

```
// GetPMUCounterRange()
// =====
// Returns the range that a counter is currently in.

PMUCounterRange GetPMUCounterRange(integer n)

    constant integer counters = NUM_PMU_COUNTERS;
    constant integer epmn = UInt(EffectiveEPMN());
    constant integer hpmn = 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();
```

## 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 PMOVSLR_EL0.

bits(64) GetPMUReadMask()
    bits(64) mask = 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';

        // Check for trap to EL2.
        if EL2Enabled() && PSTATE.EL IN {EL0, EL1} && HCR_EL2.<E2H,TGE> != '11' 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';

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
```

## 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, PMOVSCLR_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.

bits(64) GetPMUWriteMask(boolean write_counter)
    bits(64) mask = 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';
        if PMUSERENR_EL0.CR == '1' then // PMCCNTR_EL0 read-only
            mask<CYCLE\_COUNTER\_ID> = '0';
        if PMUSERENR_EL0.ER == '1' then // PMEVCNTR<n>_EL0 read-only
            mask<30:0> = Zeros(31);

    // 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';

        // Check for trap to EL2.
        if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && HCR_EL2.<E2H,TGE> != '11' then
            // If FEAT_PMuV3_ICNTR and EL2 are implemented, then so is FEAT_FGT2.
            assert IsFeatureImplemented(FEAT_FGT2);
            fgt_bit = (if write_counter then HDFGWTR2_EL2.nPMICNTR_EL0
                        else HDFGWTR2_EL2.nPMICFILTR_EL0);
            if (HaveEL\(EL3\) && SCR_EL3.FGTEn2 == '0') || fgt_bit == '0' then
                mask<INSTRUCTION\_COUNTER\_ID> = '0';

    // Traps on other counters do not affect those counters' controls in the same way.

    return mask;
```

## Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// HasElapsed64Cycles()
// =====
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.

boolean HasElapsed64Cycles();
```

## Library pseudocode for shared/debug/pmu/INSTRUCTION\_COUNTER\_ID

```
// Constant used in PMU functions to represent actions on the instruction counter.
constant integer INSTRUCTION_COUNTER_ID = 32;
```

## Library pseudocode for shared/debug/pmu/IncrementInstructionCounter

```
// IncrementInstructionCounter()
// =====
// Increment the instruction counter and possibly set overflow bits.

IncrementInstructionCounter(integer increment)
    if CountPMUEvents\(INSTRUCTION\_COUNTER\_ID\) then
        constant integer old_value = UInt(PMICNTR_EL0);
        constant integer new_value = 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.F0 = '1';
            PMOVSCLR_EL0.F0 = '1';

    return;
```

## Library pseudocode for shared/debug/pmu/IsMostSecureAccess

```
// IsMostSecureAccess()
// =====
// Returns TRUE if the security state of an access is the most secure state.

boolean IsMostSecureAccess()
    return IsMostSecureAccess\(AccessState\(\)\);

// IsMostSecureAccess()
// =====
// Returns TRUE if the security state of an access is the most secure state.

boolean IsMostSecureAccess(SecurityState access_state)
    if IsFeatureImplemented(FEAT_RME) then
        return access_state == SS\_Root;
    elsif HaveEL\(EL3\) || SecureOnlyImplementation\(\) then
        return access_state == SS\_Secure;
    else
        assert access_state == SS\_NonSecure;
        return TRUE;
```

## Library pseudocode for shared/debug/pmu/IsRange3Counter

```
// IsRange3Counter()
// =====
// Returns TRUE if the counter is in the third range.

boolean IsRange3Counter(integer n)
    return PMUCounterRange\_R3 == GetPMUCounterRange(n);
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEvent

```
// PMUCaptureEvent()
// =====
// If permitted and enabled, generate a PMU snapshot Capture event.

PMUCaptureEvent()
    assert HaveEL(EL3) && IsFeatureImplemented(FEAT_PMUv3_SS) && HaveAArch64();
    constant boolean debug_state = Halted();

    if !PMUCaptureEventAllowed() then
        // Indicate a Capture event completed, unsuccessfully
        PMSSCR_EL1.<NC,SS> = '10';
        return;
    constant integer counters = NUM_PMU_COUNTERS;
    for idx = 0 to counters - 1
        PMEVCNTRSVR_EL1[idx] = PMEVCNTR_EL0[idx];
    PMCCNTRSVR_EL1 = PMCCNTR_EL0;

    if IsFeatureImplemented(FEAT_PMUv3_ICNTR) then
        PMICNTRSVR_EL1 = PMICNTR_EL0;

    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCCTL.SS == '1' then
        if pc_sample.valid && !debug_state then
            SetPCSRActive();
            SetPCSample();
        else
            SetPCSRUnknown();

    if (IsFeatureImplemented(FEAT_BRBE) && BranchRecordAllowed(PSTATE.EL) &&
        BRBCR_EL1.FZPSS == '1' && (!HaveEL(EL2) || BRBCR_EL2.FZPSS == '1')) then
        BRBEFreeze();

    // Indicate a successful Capture event
    PMSSCR_EL1.<NC,SS> = '00';
    if !debug_state || ConstrainUnpredictableBool(Unpredictable_PMUSNAPSHOTEVENT) then
        PMUEvent(PMU_EVENT_PMU_SNAPSHOT);

    return;
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventAllowed

```
// PMUCaptureEventAllowed()
// =====
// Returns TRUE if PMU Capture events are allowed, and FALSE otherwise.

boolean PMUCaptureEventAllowed()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;

    if !PMUCaptureEventEnabled() || OSLockStatus() then
        return FALSE;
    elsif HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '11';
    elsif HaveEL(EL2) && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '11';
    else
        bits(2) pmsse_el1 = PMECR_EL1.SSE;
        if pmsse_el1 == '01' then // Reserved value
            Constrain c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMSSE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '11';
```

## Library pseudocode for shared/debug/pmu/PMUCaptureEventEnabled

```
// PMUCaptureEventEnabled()
// =====
// Returns TRUE if PMU Capture events are enabled, and FALSE otherwise.

boolean PMUCaptureEventEnabled()
    if !IsFeatureImplemented(FEAT_PMUv3_SS) || !HaveAArch64() then
        return FALSE;
    if HaveEL(EL3) && MDCR_EL3.PMSSE != '01' then
        return MDCR_EL3.PMSSE == '1x';
    elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
        return FALSE;
    elsif HaveEL(EL2) && MDCR_EL2.PMSSE != '01' then
        return MDCR_EL2.PMSSE == '1x';
    elsif ELUsingAArch32(EL1) then
        return FALSE;
    else
        bits(2) pmsse_el1 = PMECR_EL1.SSE;
        if pmsse_el1 == '01' then // Reserved value
            Constraint c;
            (c, pmsse_el1) = ConstrainUnpredictableBits(Unpredictable_RESPMSSE, 2);
            assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
            if c == Constraint_DISABLED then pmsse_el1 = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value
        return pmsse_el1 == '1x';
```



```

// 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.

integer PMUCountValue(integer n, integer Vb, integer Vm)
    assert (n MOD 2) == 1 || Vm == 0;
    assert n < NUM_PMU_COUNTERS;

    if !IsFeatureImplemented(FEAT_PMUv3_TH) || !HaveAArch64() then
        return Vb;

    constant integer TH = UInt(PMEVTYPER_EL0[n].TH);

    // Control register fields
    bits(3) tc = PMEVTYPER_EL0[n].TC;
    bit te = '0';
    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        te = PMEVTYPER_EL0[n].TE;
    bits(2) tlc = '00';
    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        tlc = PMEVTYPER_EL0[n].TLC;

    // Check for reserved cases
    Constraint c;
    (c, tc, te, tlc) = ReservedPMUThreshold(n, tc, te, tlc);
    if c == Constraint_DISABLED then
        return Vb;
    // 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;

    // Threshold condition
    boolean Ct;
    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

    integer Vn;
    if te == '1' then
        // Edge condition
        constant boolean Cp = PMULastThresholdValue[n];
        boolean Ce;
        integer Ve;
        case tc<1:0> of
            when '10' Ce = (Cp != Ct);       // Both edges
            when 'x1' Ce = (!Cp && Ct);       // Single edge
            otherwise Unreachable();          // Covered by ReservedPMUThreshold
        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
        Vn = Ve;
    else
        // Threshold condition
        integer Vt;
        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

```



```

    Vn = Vt;

    PMULastThresholdValue[n] = Ct;

    return Vn;

```

### Library pseudocode for shared/debug/pmu/PMUCounterRange

```

// PMUCounterRange
// =====
// Enumerates the ranges to which an event counter belongs to.

enumeration PMUCounterRange {
    PMUCounterRange_R1,
    PMUCounterRange_R2,
    PMUCounterRange_R3
};

```

### Library pseudocode for shared/debug/pmu/PMUEvent

```

// PMUEvent()
// =====
// Generate a PMU event. By default, increment by 1.

PMUEvent(bits(16) pmuevent)
    PMUEvent(pmuevent, 1);

// PMUEvent()
// =====
// Accumulate a PMU Event.

PMUEvent(bits(16) pmuevent, integer increment)
    if (IsFeatureImplemented(FEAT_SPE) && SPESampleInFlight) then
        SPEEvent(pmuevent);
    constant integer counters = NUM_PMU_COUNTERS;
    if counters != 0 then
        for idx = 0 to counters - 1
            PMUEvent(pmuevent, increment, idx);

    if (HaveAArch64() && IsFeatureImplemented(FEAT_PMUv3_ICNTR) &&
        pmuevent == PMU_EVENT_INST_RETIRED) then
        IncrementInstructionCounter(increment);

// PMUEvent()
// =====
// Accumulate a PMU Event for a specific event counter.

PMUEvent(bits(16) pmuevent, integer increment, integer idx)
    if !IsFeatureImplemented(FEAT_PMUv3) then
        return;

    if UsingAArch32() then
        if PMEVTYPEPER[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
    else
        if PMEVTYPEPER_ELO[idx].evtCount == pmuevent then
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;

```

## Library pseudocode for shared/debug/pmu/PMUOverflowCondition

```
// PMUOverflowCondition()
// =====
// Enumerates the reasons for which the PMU overflow condition is evaluated.

enumeration PMUOverflowCondition {
    PMUOverflowCondition_PMUException,
    PMUOverflowCondition_BRBEFreeze,
    PMUOverflowCondition_Freeze,
    PMUOverflowCondition_IRQ
};
```

## Library pseudocode for shared/debug/pmu/PMUSwIncrement

```
// PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC

PMUSwIncrement(bits(64) sw_incr_in)

    bits(64) sw_incr = sw_incr_in;
    bits(31) mask = Zeros(31);
    constant integer counters = GetNumEventCountersAccessible();
    if counters > 0 then
        mask<counters-1:0> = Ones(counters);

    if (IsFeatureImplemented(FEAT_PMUv3p9) && !ELUsingAArch32(EL1) &&
        PSTATE.EL == EL0 && PMUSERENR_EL0.<UEN,SW> == '10') then
        mask = mask AND PMUACR_EL1<30:0>;

    sw_incr = sw_incr AND ZeroExtend(mask, 64);
    for idx = 0 to 30
        if sw_incr<idx> == '1' then
            PMUEvent(PMU_EVENT_SW_INCR, 1, idx);

    return;
```

## Library pseudocode for shared/debug/pmu/ReservedPMUThreshold

```
// ReservedPMUThreshold()
// =====
// Checks if the given PMEVTYPER<n>_EL1.{TH,TE,TLC} values are reserved and will
// generate Constrained Unpredictable behavior, otherwise return Constraint_NONE.

(Constraint, bits(3), bit, bits(2)) ReservedPMUThreshold(integer n, bits(3) tc_in,
                                                         bit te_in, bits(2) tlc_in)

    bits(3) tc = tc_in;
    bit te = te_in;
    bits(2) tlc = tlc_in;

    boolean reserved = FALSE;

    if IsFeatureImplemented(FEAT_PMUv3_EDGE) then
        if te == '1' && tc<1:0> == '00' then          // Edge condition
            reserved = TRUE;
    else
        te = '0';                                     // Control is RES0

    if IsFeatureImplemented(FEAT_PMUv3_TH2) && (n MOD 2) == 1 then
        if tlc == '11' then                            // Reserved value
            reserved = TRUE;
        if te == '1' then                               // Edge condition
            if tlc == '01' then
                reserved = TRUE;
        else                                           // Threshold condition
            if tc<0> == '1' && tlc == '10' then
                reserved = TRUE;
    else
        tlc = '00';                                   // Controls are RES0

    Constraint c = Constraint\_NONE;
    if reserved then
        (c, <tc,te,tlc>) = ConstrainUnpredictableBits(Unpredictable\_RESTC, 6);

    return (c, tc, te, tlc);
```

## Library pseudocode for shared/debug/pmu/SMEPMUEventPredicate

```
// SMEPMUEventPredicate()
// =====
// Call the relevant PMU predication events based on the SME instruction properties.

SMEPMUEventPredicate(bits(N) mask1, bits(N) mask2, integer esize)
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    PMUEvent(PMU_EVENT_SME_PRED2_SPEC);
    if AllElementsActive(mask1, esize) && AllElementsActive(mask2, esize) then
        PMUEvent(PMU_EVENT_SME_PRED2_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SME_PRED2_NOT_FULL_SPEC);
        if !AnyActiveElement(mask1, esize) && !AnyActiveElement(mask2, esize) then
            PMUEvent(PMU_EVENT_SME_PRED2_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SME_PRED2_PARTIAL_SPEC);
```

## Library pseudocode for shared/debug/pmu/SVEPMUEventPredicate

```
// SVEPMUEventPredicate()
// =====
// Call the relevant PMU predication events based on the SVE instruction properties.

SVEPMUEventPredicate(bits(N) mask, integer esize)
    PMUEvent(PMU_EVENT_SVE_PRED_SPEC);
    if AllElementsActive(mask, esize) then
        PMUEvent(PMU_EVENT_SVE_PRED_FULL_SPEC);
    else
        PMUEvent(PMU_EVENT_SVE_PRED_NOT_FULL_SPEC);
        if !AnyActiveElement(mask, esize) then
            PMUEvent(PMU_EVENT_SVE_PRED_EMPTY_SPEC);
        else
            PMUEvent(PMU_EVENT_SVE_PRED_PARTIAL_SPEC);
```

## Library pseudocode for shared/debug/pmu/ShouldPMUFreeze

```
// ShouldPMUFreeze()
// =====

boolean ShouldPMUFreeze(PMUCounterRange r)
    constant boolean include_r1 = (r == PMUCounterRange\_R1);
    constant boolean include_r2 = (r == PMUCounterRange\_R2);
    constant boolean include_r3 = FALSE;

    if r == PMUCounterRange\_R3 then
        return FALSE;

    constant boolean overflow = CheckPMUOverflowCondition(PMUOverflowCondition\_Freeze,
                                                         include_r1, include_r2, include_r3);
    return overflow;
```

## 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.

ZeroCycleCounter()
    bits(64) mask = Zeros(64);
    mask<CYCLE\_COUNTER\_ID> = '1';
    ZeroPMUCounters(mask);
```

## Library pseudocode for shared/debug/pmu/ZeroPMUCounters

```
// ZeroPMUCounters()
// =====
// Zero set of counters specified by the mask in 'val'.
// For a write to PMZR_EL0, 'val' is the value passed in X<t>.

ZeroPMUCounters(bits(64) val)
    constant bits(64) masked_val = val AND GetPMUWriteMask(TRUE);

    for idx = 0 to 63
        if masked_val<idx> == '1' 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);
            otherwise
                if !HaveAArch64() then
                    PMEVCNTR[idx] = Zeros(32);
                elseif IsFeatureImplemented(FEAT_PMUv3p5) then
                    PMEVCNTR_EL0[idx] = Zeros(64);
                else
                    PMEVCNTR_EL0[idx]<31:0> = Zeros(32);

    return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/CreatePCSample

```
// CreatePCSample()
// =====

CreatePCSample()
    // 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 PCSRsuspended() then return;

    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
        if ELUsingAArch32(EL2) then
            pc_sample.vmid = ZeroExtend(VTTBR.VMID, 16);
        elseif !IsFeatureImplemented(FEAT_VMID16) || VTCR_EL2.VS == '0' then
            pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        else
            pc_sample.vmid = VTTBR_EL2.VMID;
        if ((IsFeatureImplemented(FEAT_VHE) || IsFeatureImplemented(FEAT_Debugv8p2)) &&
            !ELUsingAArch32(EL2)) then
            pc_sample.contextidr_el2 = CONTEXTIDR_EL2<31:0>;
        else
            pc_sample.contextidr_el2 = bits(32) UNKNOWN;
    pc_sample.el0h = PSTATE.EL == EL0 && IsInHost();
    return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSRsuspended

```
// PCSRsuspended()
// =====
// Returns TRUE if PC Sample-based Profiling is suspended, and FALSE otherwise.

boolean PCSRsuspended()
    if IsFeatureImplemented(FEAT_PCSRv8p9) && PMPCCTL.IMP == '1' then
        return PMPCCTL.EN == '0';
    else
        return boolean IMPLEMENTATION_DEFINED "PCSR is suspended";
```

## Library pseudocode for shared/debug/samplebasedprofiling/PCSample

```
PCSample pc_sample;

// PCSample
// =====

type PCSample is (
    boolean valid,
    bits(64) pc,
    bits(2) el,
    bit rw,
    SecurityState ss,
    boolean has_el2,
    bits(32) contextidr,
    bits(32) contextidr_el2,
    boolean el0h,
    bits(16) vmid
)
```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_EDPCSRlo

```
// Read_EDPCSRlo()
// =====

bits(32) Read_EDPCSRlo(boolean memory_mapped)
// The Software lock is OPTIONAL.
update = !memory_mapped || EDLSR.SLK == '0';           // Software locked: no side-effects
bits(32) sample;
if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
        if IsFeatureImplemented(FEAT_VHE) && EDSCR.SC2 == '1' then
            EDPCSRhi.PC = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
            EDPCSRhi.EL = pc_sample.el;
            EDPCSRhi.NS = (if pc_sample.ss == SS\_Secure then '0' else '1');
        else
            EDPCSRhi = (if pc_sample.rw == '0' then Zeros(32) else pc_sample.pc<63:32>);
            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 bits(32) UNKNOWN);
            else
                EDVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1,EL0}
                    then pc_sample.vmid else Zeros(16));
                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 PCSRhi is zero.
                // An example implementation may be "pc_sample.rw".
                EDVIDSR.HV = (if !IsZero(EDPCSRhi) then '1'
                    else bit IMPLEMENTATION_DEFINED "0 or 1");
    else
        sample = Ones(32);
        if update then
            EDPCSRhi = bits(32) UNKNOWN;
            EDCIDSR = bits(32) UNKNOWN;
            EDVIDSR = bits(32) UNKNOWN;

return sample;
```

## Library pseudocode for shared/debug/samplebasedprofiling/Read\_PMPCSR

```
// Read_PMPCSR()
// =====

bits(64) Read_PMPCSR(boolean memory_mapped)
// The Software lock is OPTIONAL.
update = !memory_mapped || PMLSR.SLK == '0';           // Software locked: no side-effects

if IsFeatureImplemented(FEAT_PCSRv8p9) && update then
    if IsFeatureImplemented(FEAT_PMuV3_SS) && PMPCSRCTL.SS == '1' then
        update = FALSE;
    elsif PMPCSRCTL.<IMP,EN> == '10' || (PMPCSRCTL.IMP == '0' && PCSRSSuspended()) then
        pc_sample.valid = FALSE;
        SetPCSRActive();

if pc_sample.valid then
    if update then SetPCSample();
    return PMPCSR;
else
    if update then SetPCSRUnknown();
    return (bits(32) UNKNOWN : Ones(32));
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRActive

```
// SetPCSRActive()
// =====
// Sets PC Sample-based Profiling to active state.

SetPCSRActive()
    if PMPCSCCTL.IMP == '1' then
        PMPCSCCTL.EN = '1';
    // If PMPCSCCTL.IMP reads as `0b0`, then PMPCSCCTL.EN is RES0, and it is
    // IMPLEMENTATION DEFINED whether PCSR is suspended or active at reset.
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSRUnknown

```
// SetPCSRUnknown()
// =====
// Sets the PC sample registers to UNKNOWN values because PC sampling
// is prohibited.

SetPCSRUnknown()
    PMPCSR<31:0> = Ones(32);
    PMPCSR<55:32> = bits(24) UNKNOWN;
    PMPCSR.EL = bits(2) UNKNOWN;
    PMPCSR.NS = bit UNKNOWN;

    PMCID1SR = bits(32) UNKNOWN;
    PMCID2SR = bits(32) UNKNOWN;

    PMVIDSR.VMID = bits(16) UNKNOWN;

    return;
```

## Library pseudocode for shared/debug/samplebasedprofiling/SetPCSample

```
// SetPCSample()
// =====
// Sets the PC sample registers to the appropriate sample values.

SetPCSample()
    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';
        else
            PMPCSR.NS = (if pc_sample.ss == SS\_Secure then '0' else '1');

    PMCID1SR = pc_sample.contextidr;
    PMCID2SR = if pc_sample.has_el2 then pc_sample.contextidr_el2 else bits(32) UNKNOWN;

    PMVIDSR.VMID = (if pc_sample.has_el2 && pc_sample.el IN {EL1, EL0} && !pc_sample.el0h
        then pc_sample.vmid else bits(16) UNKNOWN);

    return;
```



## Library pseudocode for shared/debug/softwarestep/CheckSoftwareStep

```
// CheckSoftwareStep()
// =====
// Take a Software Step exception if in the active-pending state

CheckSoftwareStep()

    // 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().
    step_enabled = (!ELUsingAArch32(DebugTarget()) && AArch64.GenerateDebugExceptions()) &&
        MDSCR_EL1.SS == '1';
    active_pending = step_enabled && PSTATE.SS == '0';    // active-pending
    if active_pending then
        AArch64.SoftwareStepException();
    ShouldAdvanceSS = TRUE;
    return;
```

## Library pseudocode for shared/debug/softwarestep/DebugExceptionReturnSS

```
// DebugExceptionReturnSS()
// =====
// Returns value to write to PSTATE.SS on an exception return or Debug state exit.

bit DebugExceptionReturnSS(bits(N) spsr)
    assert Halted() || Restarting() || PSTATE.EL != EL0;

    boolean enabled_at_source;
    if Restarting() then
        enabled_at_source = FALSE;
    elsif UsingAArch32() then
        enabled_at_source = AArch32.GenerateDebugExceptions();
    else
        enabled_at_source = AArch64.GenerateDebugExceptions();

    boolean valid;
    bits(2) dest_el;
    if IllegalExceptionReturn(spsr) then
        dest_el = PSTATE.EL;
    else
        (valid, dest_el) = ELFromSPSR(spsr);    assert valid;

    dest_ss = SecurityStateAtEL(dest_el);
    bit mask;
    boolean enabled_at_dest;
    dest_using_32 = (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
        mask = spsr<9>;
        enabled_at_dest = AArch64.GenerateDebugExceptionsFrom(dest_el, dest_ss, mask);

    ELd = DebugTargetFrom(dest_ss);
    bit SS_bit;
    if !ELUsingAArch32(ELd) && MDSCR_EL1.SS == '1' && !enabled_at_source && enabled_at_dest then
        SS_bit = spsr<21>;
    else
        SS_bit = '0';

    return SS_bit;
```

## Library pseudocode for shared/debug/softwarestep/SSAdvance

```
// SSAdvance()
// =====
// Advance the Software Step state machine.

SSAdvance()

    // 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
    // processor only needs to consider advancing the state machine from the active-not-pending
    // state.
    if !ShouldAdvanceSS then return;
    target = DebugTarget\(\);
    step_enabled = !ELUsingAArch32(target) && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    if active_not_pending then PSTATE.SS = '0';
    ShouldAdvanceSS = FALSE;
    return;
```

## Library pseudocode for shared/debug/softwarestep/SoftwareStepOpEnabled

```
// SoftwareStepOpEnabled()
// =====
// Returns a boolean indicating if execution from MDSTEPSOP_EL1 is enabled.

boolean SoftwareStepOpEnabled()

    if !IsFeatureImplemented(FEAT_STEP2) || UsingAArch32() then
        return FALSE;

    step_enabled = AArch64.GenerateDebugExceptions() && MDSCR_EL1.SS == '1';
    active_not_pending = step_enabled && PSTATE.SS == '1';
    stepop = (MDSCR_EL1.EnSTEPSOP == '1' &&
        (!HaveEL(EL3) || MDSCR_EL3.EnSTEPSOP == '1') &&
        (!EL2Enabled() || MDSCR_EL2.EnSTEPSOP == '1'));
    return active_not_pending && stepop;
```

## 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.

boolean SoftwareStep_DidNotStep();
```

## 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.

boolean SoftwareStep_SteppedEX();
```

## Library pseudocode for shared/debug/watchpoint/WatchpointInfo

```
// WatchpointInfo
// =====
// Watchpoint related fields.

type WatchpointInfo is (
    WatchpointType wptype,      // Type of watchpoint matched
    boolean maybe_false_match,  // Watchpoint matches rounded range
    integer watchpt_num,       // Matching watchpoint number
    boolean value_match        // Watchpoint match
)
)
```

## Library pseudocode for shared/debug/watchpoint/WatchpointType

```
// WatchpointType
// =====

enumeration WatchpointType {
    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.

bit EffectiveHCRX_EL2_TMEA()
    if (IsFeatureImplemented(FEAT_DoubleFault2) && EL2Enabled() &&
        !ELUsingAArch32(EL2) && IsHCRXEL2Enabled()) then
        return HCRX_EL2.TMEA;
    else
        return '0';
```

## Library pseudocode for shared/exceptions/aborts/EffectiveHCR\_AMO

```
// EffectiveHCR_AMO()
// =====
// Return the Effective value of HCR_EL2.AMO.

bit EffectiveHCR_AMO()
    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';
```

## Library pseudocode for shared/exceptions/aborts/EffectiveHCR\_TEA

```
// EffectiveHCR_TEA()
// =====
// Return the Effective value of HCR_EL2.TEA.

bit EffectiveHCR_TEA()
    if EL2Enabled() && IsFeatureImplemented(FEAT_RAS) then
        return (if ELUsingAArch32(EL2) then HCR2.TEA else HCR_EL2.TEA);
    else
        return '0';
```

## Library pseudocode for shared/exceptions/aborts/EffectiveNMEA

```
// EffectiveNMEA()
// =====
// Return the Effective value of SCR_EL3.NMEA or SCTLR2_ELx.NMEA.

bit EffectiveNMEA()
    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';
    elsif IsFeatureImplemented(FEAT_DoubleFault) && PSTATE.EL == EL3 && !UsingAArch32() then
        return SCR_EL3.NMEA AND EffectiveEA();
    else
        return '0';
```

## Library pseudocode for shared/exceptions/aborts/EffectiveSCR\_EL3\_TMEA

```
// EffectiveSCR_EL3_TMEA()
// =====
// Return the Effective value of SCR_EL3.TMEA.

bit EffectiveSCR_EL3_TMEA()
    if (IsFeatureImplemented(FEAT_DoubleFault2) && HaveEL(EL3) &&
        !ELUsingAArch32(EL3)) then
        return SCR_EL3.TMEA;
    else
        return '0';
```



```

// PhysicalErrorTarget()
// =====
// 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.

(boolean, bits(2)) PhysicalErrorTarget()
    if Halted() then
        return (TRUE, bits(2) UNKNOWN);

    constant bit effective_ea = EffectiveEA();
    constant bit effective_amo = EffectiveHCR\_AMO();
    constant bit effective_tge = EffectiveTGE();
    constant bit effective_nmea = 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.
    bit effective_aw;
    if (ELUsingAArch32(EL3) && effective_ea == '1' &&
        CurrentSecurityState() == SS\_NonSecure && effective_amo == '0') then
        effective_aw = SCR.AW;
    else
        effective_aw = '0';

    // The exception is masked by software.
    boolean masked;
    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');

    // 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 Security state indicated by
    // SCR_EL3.{NS, NSE}.
    boolean intdis = ExternalDebugInterruptsDisabled(EL1);

    bits(2) target_el = bits(2) UNKNOWN;
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        if !masked then target_el = EL3;

    elseif 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 (EffectiveSCR\_EL3\_TMEA() == '1' &&
        ((PSTATE.EL IN {EL2, EL1} && PSTATE.A == '1') ||
         (PSTATE.EL IN {EL2, EL0} && masked) ||
         (PSTATE.EL != EL3 && intdis))) then
        target_el = EL3;
        masked = FALSE;

    elseif PSTATE.EL == EL2 || IsInHost() then
        if !masked then target_el = EL2;

```

```

else
    assert (PSTATE.EL == EL1 ||
            (PSTATE.EL == EL3 && ELUsingAArch32(EL3) ||
             (PSTATE.EL == EL0 && !IsInHost()));
    if !masked then target_el = EL1;

// External debug might disable the exception for the target Exception level.
if !masked && ExternalDebugInterruptsDisabled(target_el) then
    masked = TRUE;
    target_el = bits(2) UNKNOWN;

return (masked, target_el);

```

## 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.

bits(2) SyncExternalAbortTarget(FaultRecord fault)
    constant bit effective_ea = EffectiveEA();
    constant bit effective_tea = EffectiveHCR\_TEA();
    constant bit effective_tge = EffectiveTGE();

    bits(2) target_el;
    if effective_ea == '1' || (PSTATE.EL == EL3 && !ELUsingAArch32(EL3)) then
        target_el = EL3;

    elseif (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;

    elseif EffectiveHCRX\_EL2\_TMEA() == '1' && PSTATE.A == '1' && PSTATE.EL == EL1 then
        target_el = EL2;

    elseif 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);

    return target_el;

```

## Library pseudocode for shared/exceptions/exceptions/ConditionSyndrome

```
// ConditionSyndrome()
// =====
// Return CV and COND fields of instruction syndrome

bits(5) ConditionSyndrome()

    bits(5) syndrome;

    if UsingAArch32\(\) then
        cond = AArch32.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;
        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 boolean IMPLEMENTATION_DEFINED "Condition valid for trapped T32" then
                syndrome<4> = '1';
                syndrome<3:0> = cond;
            else
                syndrome<4> = '0';
                syndrome<3:0> = bits(4) UNKNOWN;
    else
        syndrome<4> = '1';
        syndrome<3:0> = '1110';
    return syndrome;
```



## Library pseudocode for shared/exceptions/exceptions/Exception

```
// Exception
// =====
// Classes of exception.

enumeration Exception {
    Exception_Uncategorized,    // Uncategorized or unknown reason
    Exception_WFxTrap,          // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap,      // Trapped AArch32 MCR or MRC access, coproc=0b111
    Exception_CP15RRTTrap,      // Trapped AArch32 MCRR or MRRC access, coproc=0b1111
    Exception_CP14RTTTrap,      // Trapped AArch32 MCR or MRC access, coproc=0b1110
    Exception_CP14DTTTrap,      // Trapped AArch32 LDC or STC access, coproc=0b1110
    Exception_CP14RRTTrap,      // 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_LDST64BTrap,      // Trapped access to ST64BV, ST64BV0, ST64B and LD64B
    // 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_PCAlignment,       // 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_TSTARTAccessTrap,   // Trapped TSTART access
    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 is (
    Exception    exctype,    // Exception class
    IssType      syndrome,    // Syndrome record
    FullAddress  paddress,    // Physical fault address
    bits(64)    vaddress,    // Virtual fault address
    boolean     ipavalid,     // Validity of Intermediate Physical fault address
    boolean     pavalid,     // Validity of Physical fault address
    bit         NS,          // Intermediate Physical fault address space
    bits(56)    ipaddress,    // Intermediate Physical fault address
    boolean     trappedsyscallinst) // 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.

ExceptionRecord ExceptionSyndrome(Exception exceptype)

    ExceptionRecord r;

    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.NS            = '0';
    r.ipaddress      = Zeros(56);
    r.paddress.paspace = PASpace UNKNOWN;
    r.paddress.address = bits(56) UNKNOWN;
    r.trappedsyscallinst = FALSE;
    return r;
```

## Library pseudocode for shared/exceptions/traps/Undefined

```
// Undefined()
// =====

Undefined()
    if UsingAArch32() then
        AArch32.Undefined();
    else
        AArch64.Undefined();
```

## Library pseudocode for shared/functions/aborts/EncodeLDFSC

```
// EncodeLDFSC()
// =====
// Function that gives the Long-descriptor FSC code for types of Fault

bits(6) EncodeLDFSC(Fault statuscode, integer level)
    bits(6) result;

    // 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();
        return result;

    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();

        return result;
    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 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 AsyncExternal              result = '010001'; assert UsingAArch32();
        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 HWUpdateAccessFlag          result = '110001';
        when Fault Lockdown                  result = '110100'; // IMPLEMENTATION DEFINED
        when Fault Exclusive                  result = '110101'; // IMPLEMENTATION DEFINED
        otherwise                                Unreachable();

    return result;
```

## Library pseudocode for shared/functions/aborts/IPAValid

```
// IPAValid()
// =====
// Return TRUE if the IPA is reported for the abort

boolean IPAValid(FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    if fault.gpcf.gpcf != 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;
```

## Library pseudocode for shared/functions/aborts/IsAsyncAbort

```
// IsAsyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an asynchronous abort, and FALSE
// otherwise.

boolean IsAsyncAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity});

// IsAsyncAbort()
// =====

boolean IsAsyncAbort(FaultRecord fault)
    return IsAsyncAbort(fault.statuscode);
```

## Library pseudocode for shared/functions/aborts/IsDebugException

```
// IsDebugException()
// =====

boolean IsDebugException(FaultRecord fault)
    assert fault.statuscode != Fault\_None;
    return fault.statuscode == Fault\_Debug;
```

## Library pseudocode for shared/functions/aborts/IsExternalAbort

```
// IsExternalAbort()
// =====
// Returns TRUE if the abort currently being processed is an External abort and FALSE otherwise.

boolean IsExternalAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk,
        Fault\_AsyncExternal,
        Fault\_AsyncParity
    });

// IsExternalAbort()
// =====

boolean IsExternalAbort(FaultRecord fault)
    return IsExternalAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
```

## Library pseudocode for shared/functions/aborts/IsExternalSyncAbort

```
// IsExternalSyncAbort()
// =====
// Returns TRUE if the abort currently being processed is an external
// synchronous abort and FALSE otherwise.

boolean IsExternalSyncAbort(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {
        Fault\_SyncExternal,
        Fault\_SyncParity,
        Fault\_SyncExternalOnWalk,
        Fault\_SyncParityOnWalk
    });

// IsExternalSyncAbort()
// =====

boolean IsExternalSyncAbort(FaultRecord fault)
    return IsExternalSyncAbort(fault.statuscode) || fault.gpcf.gpf == GPCF\_EABT;
```

## Library pseudocode for shared/functions/aborts/IsFault

```
// IsFault()
// =====
// Return TRUE if a fault is associated with an address descriptor

boolean IsFault(AddressDescriptor addrdesc)
    return addrdesc.fault.statuscode != Fault\_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with a memory access.

boolean IsFault(Fault fault)
    return fault != Fault\_None;

// IsFault()
// =====
// Return TRUE if a fault is associated with status returned by memory.

boolean IsFault(PhysMemRetStatus retstatus)
    return retstatus.statuscode != Fault\_None;
```

## Library pseudocode for shared/functions/aborts/IsSErrorInterrupt

```
// IsSErrorInterrupt()
// =====
// Returns TRUE if the abort currently being processed is an SError interrupt, and FALSE
// otherwise.

boolean IsSErrorInterrupt(Fault statuscode)
    assert statuscode != Fault\_None;

    return (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity});

// IsSErrorInterrupt()
// =====

boolean IsSErrorInterrupt(FaultRecord fault)
    return IsSErrorInterrupt(fault.statuscode);

// Add a specific type of return value for FaultSyndrome
type IssType is (
    bits(25) iss,
    bits(24) iss2
)
```

## Library pseudocode for shared/functions/aborts/IsSecondStage

```
// IsSecondStage()
// =====

boolean IsSecondStage(FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    return fault.secondstage;
```

## Library pseudocode for shared/functions/aborts/LSInstructionSyndrome

```
// LSInstructionSyndrome()
// =====
// 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.

bits(11) LSInstructionSyndrome();
```

## 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

boolean ReportAsGPCEException(FaultRecord fault)
    assert IsFeatureImplemented(FEAT_RME);
    assert fault.statuscode IN {Fault\_GPCFOnWalk, Fault\_GPCFOnOutput};
    assert fault.gpcf.gpf != GPCF\_None;

    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;
```

## Library pseudocode for shared/functions/cache/CACHE\_OP

```
// CACHE_OP()
// =====
// Performs Cache maintenance operations as per CacheRecord.

CACHE_OP(CacheRecord cache)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/cache/CPASAtPAS

```
// CPASAtPAS()
// =====
// Get cache PA space for given PA space.

CachePASpace CPASAtPAS(PASpace pas)
    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();
```

## Library pseudocode for shared/functions/cache/CPASAtSecurityState

```
// CPASAtSecurityState()
// =====
// Get cache PA space for given security state.

CachePASpace CPASAtSecurityState(SecurityState ss)
    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;
```

## Library pseudocode for shared/functions/cache/CacheRecord

```
// CacheRecord
// =====
// Details related to a cache operation.

type CacheRecord is (
    AccessType         acctype,           // Access type
    CacheOp            cacheop,          // Cache operation
    CacheOpScope       opscope,          // Cache operation type
    CacheType          cachetype,        // Cache type
    bits(64)           regval,
    FullAddress        paddress,
    bits(64)           vaddress,          // For VA operations
    integer            setnum,            // For SW operations
    integer            waynum,            // For SW operations
    integer            level,             // For SW operations
    Shareability       shareability,
    boolean            translated,
    boolean            is_vmid_valid,     // is vmid valid for current context
    bits(16)           vmid,
    boolean            is_asid_valid,     // is asid valid for current context
    bits(16)           asid,
    SecurityState       security,
    // For cache operations to full cache or by setnum/waynum
    // For operations by address, PA space in paddress
    CachePASpace        cpas
)
```

## Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
    if opscope == CacheOpScope\_PoE then
        return FALSE;

    if opscope == CacheOpScope\_PoPA then
        return FALSE;

    if CLIDR_EL1.LoC == '000' then
        return !(boolean IMPLEMENTATION_DEFINED
            "No fault generated for DC operations if PoC is before any level of cache");

    if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope\_PoU then
        return !(boolean IMPLEMENTATION_DEFINED
            "No fault generated for DC operations if PoU is before any level of cache");

    return TRUE;
```



## Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into setnum, waynum and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
    constant integer level = UInt(regval<3:1>);
    (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.

    constant integer waybits = CeilLog2(associativity);
    constant integer setbits = CeilLog2(numsets);
    constant integer linebits = Log2(linesize);

    constant integer waynum = if associativity == 1 then 0 else UInt(regval<31:32-waybits>);
    constant integer setnum = if numsets == 1 then 0 else UInt(regval<linebits +: setbits>);

    return (setnum, waynum, level);
```

## Library pseudocode for shared/functions/cache/GetCacheInfo

```
// GetCacheInfo()
// =====
// Returns numsets, associativity & linesize in terms of actual bytes.

(integer, integer, integer) GetCacheInfo(integer level, CacheType cachetype);
```

## Library pseudocode for shared/functions/cache/ICInstNeedsTranslation

```
// ICInstNeedsTranslation()
// =====
// Check whether Instruction Cache operation needs translation.

boolean ICInstNeedsTranslation(CacheOpScope opscope)
    return boolean IMPLEMENTATION_DEFINED "Instruction Cache needs translation";
```

## Library pseudocode for shared/functions/common/ASR

```
// ASR()
// =====

bits(N) ASR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ASR\_C(x, shift);
    return result;
```

## Library pseudocode for shared/functions/common/ASR\_C

```
// ASR_C()
// =====

(bits(N), bit) ASR_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = SignExtend(x, shift+N);
    result = extended_x<(shift+N)-1:shift>;
    carry_out = extended_x<shift-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/Abs

```
// Abs()
// =====

integer Abs(integer x)
    return if x >= 0 then x else -x;

// Abs()
// =====

real Abs(real x)
    return if x >= 0.0 then x else -x;
```

### Library pseudocode for shared/functions/common/Align

```
// Align()
// =====

integer Align(integer x, integer y)
    return y * (x DIV y);

// Align()
// =====

bits(N) Align(bits(N) x, integer y)
    return Align(UInt(x), y)<N-1:0>;
```

### Library pseudocode for shared/functions/common/BitCount

```
// BitCount()
// =====

integer BitCount(bits(N) x)
    integer result = 0;
    for i = 0 to N-1
        if x<i> == '1' then
            result = result + 1;
    return result;
```

### Library pseudocode for shared/functions/common/CeilLog2

```
// CeilLog2()
// =====
// For a positive integer X, return the Log2() of X, rounded up to the next integer

integer CeilLog2(integer x)
    assert x != 0;
    return Log2(CeilPow2(x));
```

### Library pseudocode for shared/functions/common/CeilPow2

```
// CeilPow2()
// =====
// For a positive integer X, return the smallest power of 2 >= X

integer CeilPow2(integer x)
    if x == 0 then return 0;
    if x == 1 then return 1;
    return FloorPow2(x - 1) * 2;
```

### Library pseudocode for shared/functions/common/CountLeadingSignBits

```
// CountLeadingSignBits()
// =====

integer CountLeadingSignBits(bits(N) x)
    return CountLeadingZeroBits(x<N-1:1> EOR x<N-2:0>);
```

### Library pseudocode for shared/functions/common/CountLeadingZeroBits

```
// CountLeadingZeroBits()
// =====

integer CountLeadingZeroBits(bits(N) x)
    return N - (HighestSetBit(x) + 1);
```

### Library pseudocode for shared/functions/common/Elem

```
// Elem[] - non-assignment form
// =====

bits(size) Elem(bits(N) vector, integer e, integer size]
    assert e >= 0 && (e+1)*size <= N;
    return vector<(e*size+size)-1 : e*size>;

// Elem[] - assignment form
// =====

Elem(bits(N) &vector, integer e, integer size] = bits(size) value
    assert e >= 0 && (e+1)*size <= N;
    vector<(e+1)*size-1:e*size> = value;
    return;
```

### Library pseudocode for shared/functions/common/Extend

```
// Extend()
// =====

bits(N) Extend(bits(M) x, integer N, boolean unsigned)
    return if unsigned then ZeroExtend(x, N) else SignExtend(x, N);
```

### Library pseudocode for shared/functions/common/FloorPow2

```
// FloorPow2()
// =====
// For a positive integer X, return the largest power of 2 <= X

integer FloorPow2(integer x)
    assert x >= 0;
    integer n = 1;
    if x == 0 then return 0;
    while x >= 2^n do
        n = n + 1;
    return 2^(n - 1);
```

### Library pseudocode for shared/functions/common/HighestSetBit

```
// HighestSetBit()
// =====

integer HighestSetBit(bits(N) x)
    for i = N-1 downto 0
        if x<i> == '1' then return i;
    return -1;
```

## Library pseudocode for shared/functions/common/HighestSetBitNZ

```
// HighestSetBitNZ
// =====
// Position of the highest 1 bit in a bitvector.
// Asserts if the bitvector is entirely zero.

integer HighestSetBitNZ(bits(N) x)
    assert !IsZero(x);
    return HighestSetBit(x);
```

## Library pseudocode for shared/functions/common/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    return if unsigned then UInt(x) else SInt(x);
```

## Library pseudocode for shared/functions/common/IsAligned

```
// IsAligned()
// =====

boolean IsAligned(bits(N) x, integer y)
    return x == Align(x, y);
```

## Library pseudocode for shared/functions/common/IsEven

```
// IsEven()
// =====

boolean IsEven(integer val)
    return val MOD 2 == 0;
```

## Library pseudocode for shared/functions/common/IsOdd

```
// IsOdd()
// =====

boolean IsOdd(integer val)
    return val MOD 2 == 1;
```

## Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

## Library pseudocode for shared/functions/common/IsPow2

```
// IsPow2()
// =====
// Return TRUE if integer X is positive and a power of 2. Otherwise,
// return FALSE.

boolean IsPow2(integer x)
    if x <= 0 then return FALSE;
    return FloorPow2(x) == CeilPow2(x);
```

### Library pseudocode for shared/functions/common/IsZero

```
// IsZero()
// =====

boolean IsZero(bits(N) x)
    return x == Zeros(N);
```

### Library pseudocode for shared/functions/common/IsZeroBit

```
// IsZeroBit()
// =====

bit IsZeroBit(bits(N) x)
    return if IsZero(x) then '1' else '0';
```

### Library pseudocode for shared/functions/common/LSL

```
// LSL()
// =====

bits(N) LSL(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSL\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/LSL\_C

```
// LSL_C()
// =====

(bits(N), bit) LSL_C(bits(N) x, integer shift)
    assert shift > 0 && shift < 256;
    extended_x = x : Zeros(shift);
    result = extended_x<N-1:0>;
    carry_out = extended_x<N>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LSR

```
// LSR()
// =====

bits(N) LSR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = LSR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/LSR\_C

```
// LSR_C()
// =====

(bits(N), bit) LSR_C(bits(N) x, integer shift)
  assert shift > 0 && shift < 256;
  extended_x = ZeroExtend(x, shift+N);
  result = extended_x<(shift+N)-1:shift>;
  carry_out = extended_x<shift-1>;
  return (result, carry_out);
```

### Library pseudocode for shared/functions/common/LowestSetBit

```
// LowestSetBit()
// =====

integer LowestSetBit(bits(N) x)
  for i = 0 to N-1
    if x<i> == '1' then return i;
  return N;
```

### Library pseudocode for shared/functions/common/LowestSetBitNZ

```
// LowestSetBitNZ
// =====
// Position of the lowest 1 bit in a bitvector.
// Asserts if the bit-vector is entirely zero.

integer LowestSetBitNZ(bits(N) x)
  assert !IsZero(x);
  return LowestSetBit(x);
```

### Library pseudocode for shared/functions/common/Max

```
// Max()
// =====

integer Max(integer a, integer b)
  return if a >= b then a else b;

// Max()
// =====

real Max(real a, real b)
  return if a >= b then a else b;
```

### Library pseudocode for shared/functions/common/Min

```
// Min()
// =====

integer Min(integer a, integer b)
  return if a <= b then a else b;

// Min()
// =====

real Min(real a, real b)
  return if a <= b then a else b;
```

### Library pseudocode for shared/functions/common/NormalizeReal

```
// NormalizeReal
// =====
// Normalizes x to the form 1.xxx... x 2^y and returns (mantissa, exponent)

(real, integer) NormalizeReal(real x)
    real mantissa = x;
    integer exponent = 0;
    while mantissa < 1.0 do
        mantissa = mantissa * 2.0;  exponent = exponent - 1;
    while mantissa >= 2.0 do
        mantissa = mantissa / 2.0;  exponent = exponent + 1;
    return (mantissa, exponent);
```

### Library pseudocode for shared/functions/common/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);
```

### Library pseudocode for shared/functions/common/ROR

```
// ROR()
// =====

bits(N) ROR(bits(N) x, integer shift)
    assert shift >= 0;
    bits(N) result;
    if shift == 0 then
        result = x;
    else
        (result, -) = ROR\_C(x, shift);
    return result;
```

### Library pseudocode for shared/functions/common/ROR\_C

```
// ROR_C()
// =====

(bits(N), bit) ROR_C(bits(N) x, integer shift)
    assert shift != 0 && shift < 256;
    m = shift MOD N;
    result = LSR(x,m) OR LSL(x,N-m);
    carry_out = result<N-1>;
    return (result, carry_out);
```

### Library pseudocode for shared/functions/common/RShr

```
// RShr()
// =====
// Shift integer value right with rounding

integer RShr(integer value, integer shift, boolean round)
    assert shift > 0;
    if round then
        return (value + (1 << (shift - 1))) >> shift;
    else
        return value >> shift;
```

### Library pseudocode for shared/functions/common/Replicate

```
// Replicate()
// =====

bits(M*N) Replicate(bits(M) x, integer N);
```

### Library pseudocode for shared/functions/common/Reverse

```
// Reverse()
// =====
// Reverse subwords of M bits in an N-bit word

bits(N) Reverse(bits(N) word, integer M)
    assert M > 0;
    assert N MOD M == 0;
    bits(N) result;
    constant integer swsize = M;
    constant integer sw = N DIV swsize;
    for s = 0 to sw-1
        Elem[result, (sw - 1) - s, swsize] = Elem[word, s, swsize];
    return result;
```

### Library pseudocode for shared/functions/common/RoundDown

```
// RoundDown()
// =====

integer RoundDown(real x);
```

### Library pseudocode for shared/functions/common/RoundTowardsZero

```
// RoundTowardsZero()
// =====

integer RoundTowardsZero(real x)
    return if x == 0.0 then 0 else if x >= 0.0 then RoundDown(x) else RoundUp(x);
```

### Library pseudocode for shared/functions/common/RoundUp

```
// RoundUp()
// =====

integer RoundUp(real x);
```

### Library pseudocode for shared/functions/common/SInt

```
// SInt()
// =====

integer SInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    if x<N-1> == '1' then result = result - 2^N;
    return result;
```

### Library pseudocode for shared/functions/common/SignExtend

```
// SignExtend()
// =====

bits(N) SignExtend(bits(M) x, integer N)
    assert N >= M;
    return Replicate(x<M-1>, N-M) : x;
```



### Library pseudocode for shared/functions/common/Signal

```
// Signal
// =====
// Available signal types

enumeration Signal {Signal_Low, Signal_High};
```

### Library pseudocode for shared/functions/common/Split

```
// Split()
// =====

(bits(M-N), bits(N)) Split(bits(M) value, integer N)
    assert M > N;
    return (value<M-1:N>, value<N-1:0>);
```

### Library pseudocode for shared/functions/common/UInt

```
// UInt()
// =====

integer UInt(bits(N) x)
    result = 0;
    for i = 0 to N-1
        if x<i> == '1' then result = result + 2^i;
    return result;
```

### Library pseudocode for shared/functions/common/ZeroExtend

```
// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x, integer N)
    assert N >= M;
    return Zeros(N-M) : x;
```

### Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0',N);
```

## Library pseudocode for shared/functions/counters/AArch32.CheckTimerConditions

```
// AArch32.CheckTimerConditions()
// =====
// Checking timer conditions for all A32 timer registers

AArch32.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    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';

        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';
    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';

    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';

    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';

    return;
```

## Library pseudocode for shared/functions/counters/AArch64.CheckTimerConditions

```
// AArch64.CheckTimerConditions()
// =====
// Checking timer conditions for all A64 timer registers

AArch64.CheckTimerConditions()
    boolean status;
    bits(64) offset;
    bit imask;
    constant SecurityState ss = 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);
    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';
        status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                     imask, InterruptID CNTP);
        CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
    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';
    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';

    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';

    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';
        status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                     imask, InterruptID CNTV);
        CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

    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';

    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';
    return;
```

## Library pseudocode for shared/functions/counters/CNTHCTL\_EL2\_VHE

```
// CNTHCTL_EL2_VHE()
// =====
// In the case where EL2 accesses the CNTKCTL_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 CNTKCTL_EL1 are
// treated as being UNKNOWN. This function applies the UNKNOWN behavior.

bits(64) CNTHCTL_EL2_VHE(bits(64) original_value)
    assert PSTATE.EL == EL2;
    assert HCR_EL2.E2H == '1';

    bits(64) return_value = original_value;
    if !IsFeatureImplemented(FEAT_NV2p1) then
        return_value<19:18> = bits(2) UNKNOWN;
        return_value<16:10> = bits(7) UNKNOWN;
    return return_value;
```

## Library pseudocode for shared/functions/counters/GenericCounterTick

```
// GenericCounterTick()
// =====
// Increments PhysicalCount value for every clock tick.

GenericCounterTick()
    bits(64) prev_physical_count;
    if CNTCR.EN == '0' then
        if !HaveAArch64() then
            AArch32.CheckTimerConditions();
        else
            AArch64.CheckTimerConditions();
        return;
    prev_physical_count = PhysicalCountInt();
    if IsFeatureImplemented(FEAT_CNTSC) && CNTCR.SCEN == '1' then
        PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR, 88);
    else
        PhysicalCount<87:24> = PhysicalCount<87:24> + 1;
    if !HaveAArch64() then
        AArch32.CheckTimerConditions();
    else
        AArch64.CheckTimerConditions();
    TestEventCNTP(prev_physical_count, PhysicalCountInt());
    TestEventCNTV(prev_physical_count, PhysicalCountInt());
    return;
```

## Library pseudocode for shared/functions/counters/IsLocalTimeoutEventPending

```
boolean IsLocalTimeoutEventPending;
```

## Library pseudocode for shared/functions/counters/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                           bits(1) imask, InterruptID intid)
    boolean condition_met;
    Signal level;
    condition_met = (UInt(PhysicalCountInt() - offset) - UInt(compare_value)) >= 0;
    level = if condition_met && imask == '0' then Signal\_High else Signal\_Low;
    SetInterruptRequestLevel(intid, level);
    return condition_met;
```

## Library pseudocode for shared/functions/counters/LocalTimeoutVal

```
bits(64) LocalTimeoutVal; // Value to compare against the Virtual Counter Timer
```

## Library pseudocode for shared/functions/counters/PhysicalCount

```
bits(88) PhysicalCount;
```

## Library pseudocode for shared/functions/counters/SetEventRegister

```
// SetEventRegister()
// =====
// Sets the Event Register of this PE

SetEventRegister()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTP

```
// TestEventCNTP()
// =====
// Generate Event stream from the physical counter

TestEventCNTP(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    if CNTHCTL_EL2.EVNTEN == '1' then
        n = UInt(CNTHCTL_EL2.EVNTI);
        if IsFeatureImplemented(FEAT_ECV) && CNTHCTL_EL2.EVNTIS == '1' then
            n = n + 8;
        if (IsFeatureImplemented(FEAT_ECV_POFF) && EL2Enabled() && !ELIsInHost(EL0) &&
            CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1') then
            offset = CNTPOFF_EL2;
        else
            offset = Zeros(64);
        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();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

## Library pseudocode for shared/functions/counters/TestEventCNTV

```
// TestEventCNTV()
// =====
// Generate Event stream from the virtual counter

TestEventCNTV(bits(64) prev_physical_count, bits(64) current_physical_count)
    bits(64) offset;
    bits(1) samplebit, previousbit;
    integer n;
    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;
        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();
        else
            if previousbit == '1' && samplebit == '0' then SetEventRegister();
    return;
```

## 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.

bits(64) VirtualCounterTimer()
    bits(64) cntvct;

    if PSTATE.EL != EL3 then
        if HaveEL(EL2) && !ELIsInHost(PSTATE.EL) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2;
        else
            cntvct = PhysicalCountInt();
    else
        if HaveEL(EL2) && !ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF_EL2;
        elsif HaveEL(EL2) && ELUsingAArch32(EL2) then
            cntvct = PhysicalCountInt() - CNTVOFF;
        else
            cntvct = PhysicalCountInt();

    return cntvct;
```

## Library pseudocode for shared/functions/crc/BitReverse

```
// BitReverse()
// =====

bits(N) BitReverse(bits(N) data)
    bits(N) result;
    for i = 0 to N-1
        result<(N-i)-1> = data<i>;
    return result;
```

## Library pseudocode for shared/functions/crc/Poly32Mod2

```
// Poly32Mod2()
// =====

// Poly32Mod2 on a bitstring does a polynomial Modulus over {0,1} operation

bits(32) Poly32Mod2(bits(N) data_in, bits(32) poly)
    assert N > 32;
    bits(N) data = data_in;
    for i = N-1 downto 32
        if data<i> == '1' then
            data<i-1:0> = data<i-1:0> EOR (poly:Zeros(i-32));
    return data<31:0>;
```

## Library pseudocode for shared/functions/crypto/AESInvMixColumns

```
// AESInvMixColumns()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESMixColumns.

bits(128) AESInvMixColumns(bits (128) op)
    constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

    bits(4*8) out0;
    bits(4*8) out1;
    bits(4*8) out2;
    bits(4*8) out3;

    for c = 0 to 3
        out0<c*8+:8> = (FFmul0E(in0<c*8+:8>) EOR FFmul0B(in1<c*8+:8>) EOR FFmul0D(in2<c*8+:8>) EOR
                        FFmul09(in3<c*8+:8>));
        out1<c*8+:8> = (FFmul09(in0<c*8+:8>) EOR FFmul0E(in1<c*8+:8>) EOR FFmul0B(in2<c*8+:8>) EOR
                        FFmul0D(in3<c*8+:8>));
        out2<c*8+:8> = (FFmul0D(in0<c*8+:8>) EOR FFmul09(in1<c*8+:8>) EOR FFmul0E(in2<c*8+:8>) EOR
                        FFmul0B(in3<c*8+:8>));
        out3<c*8+:8> = (FFmul0B(in0<c*8+:8>) EOR FFmul0D(in1<c*8+:8>) EOR FFmul09(in2<c*8+:8>) EOR
                        FFmul0E(in3<c*8+:8>));

    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>
    );
```

## Library pseudocode for shared/functions/crypto/AESInvShiftRows

```
// AESInvShiftRows()
// =====
// Transformation in the Inverse Cipher that is inverse of AESShiftRows.

bits(128) AESInvShiftRows(bits(128) op)
    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>
    );
```

## Library pseudocode for shared/functions/crypto/AESInvSubBytes

```
// AESInvSubBytes()
// =====
// Transformation in the Inverse Cipher that is the inverse of AESSubBytes.

bits(128) AESInvSubBytes(bits(128) op)
  // Inverse S-box values
  constant bits(16*16*8) GF2_inv = (
    /*      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*/ 0x6b8a130103bdafc1020f3fca8f1e2cd0<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>
  );
  bits(128) out;
  for i = 0 to 15
    out<i*8+:8> = GF2_inv<UInt>(op<i*8+:8>)*8+:8>;
  return out;
```

## 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.

bits(128) AESMixColumns(bits(128) op)
  constant bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  constant bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  constant bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  constant bits(4*8) in3 = op<120+:8> : op< 88+:8> : op< 56+:8> : op< 24+:8>;

  bits(4*8) out0;
  bits(4*8) out1;
  bits(4*8) out2;
  bits(4*8) out3;

  for c = 0 to 3
    out0<c*8+:8> = (FFmul02(in0<c*8+:8>) EOR FFmul03(in1<c*8+:8>) EOR
                    in2<c*8+:8> EOR in3<c*8+:8>);
    out1<c*8+:8> = (FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
                    in3<c*8+:8> EOR in0<c*8+:8>);
    out2<c*8+:8> = (FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>) EOR
                    in0<c*8+:8> EOR in1<c*8+:8>);
    out3<c*8+:8> = (FFmul02(in3<c*8+:8>) EOR FFmul03(in0<c*8+:8>) EOR
                    in1<c*8+:8> EOR in2<c*8+:8>);

  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>
  );
```



## 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.

bits(128) AESShiftRows(bits(128) op)
    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>
    );
```

## 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.

bits(128) AESSubBytes(bits(128) op)
    // S-box values
    constant bits(16*16*8) GF2 = (
        /*
         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*/ 0xc072a49cafa2d4adf04759fa7dc982ca<127:0> :
        /*0*/ 0x76abd7fe2b670130c56f6bf27b777c63<127:0>
    );
    bits(128) out;
    for i = 0 to 15
        out<i*8+:8> = GF2<UInt
```

## Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
    constant bits(256*8) FFmul_02 = (
        /*      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*/ 0x252721232D2F292B353731333D3F393B<127:0> :
        /*8*/ 0x050701030D0F090B151711131D1F191B<127:0> :
        /*7*/ 0xFEFCFAF8F6F4F2F0EEECFAE8E6E4E2E0<127:0> :
        /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0<127:0> :
        /*5*/ 0xBEBBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
        /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
        /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
        /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
        /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
        /*0*/ 0x1E1C1A1816141210E0C0A0806040200<127:0>
    );
    return FFmul_02<UInt>(b)*8+:8;
```

## Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
    constant bits(256*8) FFmul_03 = (
        /*      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*/ 0xD1D2D7D4DDDEDBD8C9CACFCCC5C6C3C0<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;
```

## Library pseudocode for shared/functions/crypto/FFmul09

```
// FFmul09()
// =====

bits(8) FFmul09(bits(8) b)
    constant bits(256*8) FFmul_09 = (
        /*      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;
```

## Library pseudocode for shared/functions/crypto/FFmul0B

```
// FFmul0B()
// =====

bits(8) FFmul0B(bits(8) b)
    constant bits(256*8) FFmul_0B = (
        /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
        /*F*/ 0xA3A8B5BE8F849992F8F0EDE6D7DCC1CA<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;
```

## Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
    constant bits(256*8) FFmul_0D = (
        /*      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*/ 0xF6FBCECE1C2CFD8D59E938489AAA7B0BD<127:0> :
        /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
        /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
        /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
        /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADDD0<127:0> :
        /*0*/ 0x4B46515C7F726568232E3934171A0D00<127:0>
    );
    return FFmul_0D<UInt>(b)*8+:8>;
```

## Library pseudocode for shared/functions/crypto/FFmul0E

```
// FFmul0E()
// =====

bits(8) FFmul0E(bits(8) b)
    constant bits(256*8) FFmul_0E = (
        /*      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*/ 0x202E3C321816040A505E4C426866747A<127:0> :
        /*A*/ 0xC0CEDCD2F8F6E4EAB0BEACA28886949A<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>;
```

## Library pseudocode for shared/functions/crypto/ROL

```
// ROL()
// =====

bits(N) ROL(bits(N) x, integer shift)
    assert shift >= 0 && shift <= N;
    if (shift == 0) then
        return x;
    return ROR(x, N-shift);
```

### Library pseudocode for shared/functions/crypto/SHA256hash

```
// SHA256hash()
// =====

bits(128) SHA256hash(bits (128) x_in, bits(128) y_in, bits(128) w, boolean part1)
  bits(32) chs, maj, t;
  bits(128) x = x_in;
  bits(128) y = y_in;

  for e = 0 to 3
    chs = SHAchoose(y<31:0>, y<63:32>, y<95:64>);
    maj = SHAmajority(x<31:0>, x<63:32>, x<95:64>);
    t = y<127:96> + SHAhashSIGMA1(y<31:0>) + chs + Elem[w, e, 32];
    x<127:96> = t + x<127:96>;
    y<127:96> = t + SHAhashSIGMA0(x<31:0>) + maj;
    <y, x> = ROL(y : x, 32);
  return (if part1 then x else y);
```

### Library pseudocode for shared/functions/crypto/SHAchoose

```
// SHAchoose()
// =====

bits(32) SHAchoose(bits(32) x, bits(32) y, bits(32) z)
  return ((y EOR z) AND x) EOR z;
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA0

```
// SHAhashSIGMA0()
// =====

bits(32) SHAhashSIGMA0(bits(32) x)
  return ROR(x, 2) EOR ROR(x, 13) EOR ROR(x, 22);
```

### Library pseudocode for shared/functions/crypto/SHAhashSIGMA1

```
// SHAhashSIGMA1()
// =====

bits(32) SHAhashSIGMA1(bits(32) x)
  return ROR(x, 6) EOR ROR(x, 11) EOR ROR(x, 25);
```

### Library pseudocode for shared/functions/crypto/SHAmajority

```
// SHAmajority()
// =====

bits(32) SHAmajority(bits(32) x, bits(32) y, bits(32) z)
  return ((x AND y) OR ((x OR y) AND z));
```

### Library pseudocode for shared/functions/crypto/SHAparity

```
// SHAparity()
// =====

bits(32) SHAparity(bits(32) x, bits(32) y, bits(32) z)
  return (x EOR y EOR z);
```

## Library pseudocode for shared/functions/crypto/Sbox

```
// Sbox()
// =====
// Used in SM4E crypto instruction

bits(8) Sbox(bits(8) sboxin)
    bits(8) sboxout;
    constant bits(2048) sboxstring = (
        /*      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>
    );
    constant integer sboxindex = 255 - UInt(sboxin);
    sboxout = Elem[sboxstring, sboxindex, 8];
    return sboxout;
```

## Library pseudocode for shared/functions/decode/DecodeType

```
// DecodeType
// =====

enumeration DecodeType {
    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.

EndOfDecode(DecodeType reason)
    if IsFeatureImplemented(FEAT_BTI) && !UsingAArch32() then
        BranchTargetCheck();
    case reason of
        when Decode\_NOP    ExecuteAsNOP();
        when Decode\_UNDEF Undefined();
        when Decode\_OK      // Continue to execute.
    return;
```

## 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 address.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.

ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// ClearExclusiveLocal()
// =====
// Clear the local Exclusives monitor for the specified processorid.

ClearExclusiveLocal(integer processorid);
```

### 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.

bit ExclusiveMonitorsStatus();
```

### 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 address.

boolean IsExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### 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 address.

boolean IsExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// MarkExclusiveGlobal()
// =====
// Record the physical address region of size bytes starting at address in
// the global Exclusives monitor for processorid.

MarkExclusiveGlobal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// MarkExclusiveLocal()
// =====
// Record the physical address region of size bytes starting at address in
// the local Exclusives monitor for processorid.

MarkExclusiveLocal(FullAddress paddress, integer processorid, integer size);
```

### Library pseudocode for shared/functions/exclusive/ProcessorID

```
// ProcessorID()
// =====
// Return the ID of the currently executing PE.

integer ProcessorID();
```

## Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if IsFeatureImplemented(FEAT_Debugv8p4) then
        return FALSE;
    if IsFeatureImplemented(FEAT_DoPD) && component != Component_CTI then
        return FALSE;
    case component of
        when Component_ETE
            return boolean IMPLEMENTATION_DEFINED "ETE has Software Lock";
        when Component_Debug
            return boolean IMPLEMENTATION_DEFINED "Debug has Software Lock";
        when Component_PMU
            return boolean IMPLEMENTATION_DEFINED "PMU has Software Lock";
        when Component_CTI
            return boolean IMPLEMENTATION_DEFINED "CTI has Software Lock";
        otherwise
            Unreachable();
```

## Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return IsFeatureImplemented(FEAT_ETE) || IsFeatureImplemented(FEAT_ETMv4);
```

## 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.

boolean InsertIESBBeforeException(bits(2) el)
    return (IsFeatureImplemented(FEAT_IESB) && boolean IMPLEMENTATION_DEFINED
        "Has Implicit Error Synchronization Barrier before Exception");
```

## Library pseudocode for shared/functions/externalaborts/ActionRequired

```
// ActionRequired()
// =====
// Return an implementation specific value:
// returns TRUE if action is required, FALSE otherwise.

boolean ActionRequired();
```

## Library pseudocode for shared/functions/externalaborts/ClearPendingDelegatedSError

```
// ClearPendingDelegatedSError()
// =====
// Clear a pending delegated SError interrupt.

ClearPendingDelegatedSError()
    assert IsFeatureImplemented(FEAT_E3DSE);
    SCR_EL3.DSE = '0';
```



### Library pseudocode for shared/functions/externalaborts/ClearPendingPhysicalSError

```
// ClearPendingPhysicalSError()
// =====
// Clear a pending physical SError interrupt.

ClearPendingPhysicalSError();
```

### Library pseudocode for shared/functions/externalaborts/ClearPendingVirtualSError

```
// ClearPendingVirtualSError()
// =====
// Clear a pending virtual SError interrupt.

ClearPendingVirtualSError()
    if ELUsingAArch32(EL2) then
        HCR.VA = '0';
    else
        HCR_EL2.VSE = '0';
```

### Library pseudocode for shared/functions/externalaborts/ErrorIsContained

```
// ErrorIsContained()
// =====
// Return an implementation specific value:
// TRUE if Error is contained by the PE, FALSE otherwise.

boolean ErrorIsContained();
```

### 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.

boolean ErrorIsSynchronized();
```

### Library pseudocode for shared/functions/externalaborts/ExtAbortToAArch64

```
// ExtAbortToAArch64()
// =====
// Returns TRUE if synchronous exception is being taken to an Exception level using AArch64.

boolean ExtAbortToAArch64(FaultRecord fault)
    assert IsExternalSyncAbort(fault.statuscode);

    return !ELUsingAArch32(SyncExternalAbortTarget(fault));
```

### Library pseudocode for shared/functions/externalaborts/FaultIsCorrected

```
// FaultIsCorrected()
// =====
// Return an implementation specific value:
// TRUE if fault is corrected by the PE, FALSE otherwise.

boolean FaultIsCorrected();
```

## Library pseudocode for shared/functions/externalaborts/GetPendingPhysicalSError

```
// GetPendingPhysicalSError()
// =====
// Returns the FaultRecord containing details of pending Physical SError
// interrupt.

FaultRecord GetPendingPhysicalSError();
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalAbort

```
// HandleExternalAbort()
// =====
// Takes a Synchronous/Asynchronous abort based on fault.

HandleExternalAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                   AddressDescriptor memaddrdesc, integer size,
                   AccessDescriptor accdesc)
    assert (memretstatus.statuscode IN {Fault\_SyncExternal, Fault\_AsyncExternal} ||
            (!IsFeatureImplemented(FEAT_RAS) && memretstatus.statuscode IN {Fault\_SyncParity,
                                                                              Fault\_AsyncParity}));

    fault = 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;

    if IsFeatureImplemented(FEAT_RAS) then
        fault.merrorstate = memretstatus.merrorstate;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32() then
            AArch32.Abort(fault);
        else
            AArch64.Abort(fault);

    else
        PendSErrorInterrupt(fault);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalReadAbort

```
// HandleExternalReadAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory read.

HandleExternalReadAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                       integer size, AccessDescriptor accdesc)
    iswrite = FALSE;
    HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalTTWAbort

```
// HandleExternalTTWAbort()
// =====
// Take Asynchronous abort or update FaultRecord for Translation Table Walk
// based on PhysMemRetStatus.

FaultRecord HandleExternalTTWAbort(PhysMemRetStatus memretstatus, boolean iswrite,
                                   AddressDescriptor memaddrdesc,
                                   AccessDescriptor accdesc, integer size,
                                   FaultRecord input_fault)

output_fault = 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;

// 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;
if IsFeatureImplemented(FEAT_RAS) then
    output_fault.merrorstate = memretstatus.merrorstate;
if !IsExternalSyncAbort(output_fault) then
    PendSErrorInterrupt(output_fault);
    output_fault.statuscode = Fault\_None;
return output_fault;
```

## Library pseudocode for shared/functions/externalaborts/HandleExternalWriteAbort

```
// HandleExternalWriteAbort()
// =====
// Wrapper function for HandleExternalAbort function in case of an External
// Abort on memory write.

HandleExternalWriteAbort(PhysMemRetStatus memstatus, AddressDescriptor memaddrdesc,
                        integer size, AccessDescriptor accdesc)

iswrite = TRUE;
HandleExternalAbort(memstatus, iswrite, memaddrdesc, size, accdesc);
```

## 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.

boolean IsExternalAbortTakenSynchronously(PhysMemRetStatus memstatus, boolean iswrite,
AddressDescriptor desc, integer size,
AccessDescriptor accdesc);
```

## Library pseudocode for shared/functions/externalaborts/IsPhysicalSErrorPending

```
// IsPhysicalSErrorPending()
// =====
// Returns TRUE if a physical SError interrupt is pending.

boolean IsPhysicalSErrorPending();
```

## Library pseudocode for shared/functions/externalaborts/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered()
    if IsFeatureImplemented(FEAT_DoubleFault) then
        return TRUE;
    else
        return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

## Library pseudocode for shared/functions/externalaborts/IsSynchronizablePhysicalSErrorPending

```
// IsSynchronizablePhysicalSErrorPending()
// =====
// Returns TRUE if a synchronizable physical SError interrupt is pending.

boolean IsSynchronizablePhysicalSErrorPending();
```

## Library pseudocode for shared/functions/externalaborts/IsVirtualSErrorPending

```
// IsVirtualSErrorPending()
// =====
// Return TRUE if a virtual SError interrupt is pending.

boolean IsVirtualSErrorPending()
    if ELUsingAArch32(EL2) then
        return HCR.VA == '1';
    else
        return HCR_EL2.VSE == '1';
```

## Library pseudocode for shared/functions/externalaborts/PEErrorState

```
// PErrorState()
// =====
// 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.

ErrorState PErrorState(FaultRecord fault)
    assert !FaultIsCorrected();
    if (!ErrorIsContained() ||
        (!ErrorIsSynchronized() && !StateIsRecoverable()) ||
        ReportErrorAsUC()) then
        return ErrorState_UC;

    if !StateIsRecoverable() || ReportErrorAsUEU() then
        return ErrorState_UEU;

    if ActionRequired() || ReportErrorAsUER() then
        return ErrorState_UER;

    return ErrorState_UEO;
```

## Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// PendSErrorInterrupt()
// =====
// Pend the SError Interrupt.

PendSErrorInterrupt(FaultRecord fault);
```

## Library pseudocode for shared/functions/externalaborts/ReportErrorAsIMPDEF

```
// ReportErrorAsIMPDEF()
// =====
// Return an implementation specific value:
// returns TRUE if Error is IMPDEF, FALSE otherwise.

boolean ReportErrorAsIMPDEF();
```

## Library pseudocode for shared/functions/externalaborts/ReportErrorAsUC

```
// ReportErrorAsUC()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Uncontainable, FALSE otherwise.

boolean ReportErrorAsUC();
```

## Library pseudocode for shared/functions/externalaborts/ReportErrorAsUER

```
// ReportErrorAsUER()
// =====
// Return an implementation specific value:
// returns TRUE if Error is Recoverable, FALSE otherwise.

boolean ReportErrorAsUER();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUEU

```
// ReportErrorAsUEU()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is Unrecoverable, FALSE otherwise.  
  
boolean ReportErrorAsUEU();
```

### Library pseudocode for shared/functions/externalaborts/ReportErrorAsUncategorized

```
// ReportErrorAsUncategorized()  
// =====  
// Return an implementation specific value:  
// returns TRUE if Error is uncategorized, FALSE otherwise.  
  
boolean ReportErrorAsUncategorized();
```

### Library pseudocode for shared/functions/externalaborts/StateIsRecoverable

```
// StateIsRecoverable()  
// =====  
// Return an implementation specific value:  
// returns TRUE is PE State is unrecoverable else FALSE.  
  
boolean StateIsRecoverable();
```

## Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =====
// Non-widening BFloat16 addition used by SVE2 instructions.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFAdd(op1, op2, fpcr, fpexc);

// 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.

bits(N) BFAdd(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

            if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFAdd\_ZA

```
// BFAdd_ZA()
// =====
// Non-widening BFloat16 addition used by SME2 ZA-targeting instructions.

bits(N) BFAdd_ZA(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    constant boolean fpexc = FALSE;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFAdd(op1, op2, fpcr, fpexc);
```

## 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

bits(32) BFDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCR\_Type fpcr_in)
FPCR\_Type fpcr = fpcr_in;
bits(32) prod;

bits(32) result;
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
    constant boolean isbfloat16 = TRUE;
    constant boolean fpexc = 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(addend, prod, fpcr, fpexc);

return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFInfinity

```
// BFInfinity()
// =====

bits(N) BFInfinity(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Ones(E) : Zeros(F);
```

## Library pseudocode for shared/functions/float/bfloat/BFMatMulAdd

```
// BFMatMulAdd()
// =====
// BFloat16 matrix multiply and add to single-precision matrix
// result[2, 2] = addend[2, 2] + (op1[2, 4] * op2[4, 2])

bits(N) BFMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N == 128;

bits(N) result;
bits(32) sum;

for i = 0 to 1
    for j = 0 to 1
        sum = Elem[addend, 2*i + j, 32];
        for k = 0 to 1
            constant bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
            constant bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
            constant bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
            constant bits(16) elt2_b = Elem[op2, 4*j + 2*k + 1, 16];
            sum = BFDotAdd(sum, elt1_a, elt1_b, elt2_a, elt2_b, fpcr);
        Elem[result, 2*i + j, 32] = sum;

return result;
```



## Library pseudocode for shared/functions/float/bfloat/BFMax

```
// BFMax()
// =====
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return BFMax(op1, op2, fpcr, altfp, fpexc);

// BFMax()
// =====
// BFloat16 maximum.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return BFMax(op1, op2, fpcr, altfp, fpexc);

// 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.

bits(N) BFMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N == 16;
    FPCR\_Type fpcr = fpcr_in;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    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);
        return (if type2 == FPTYPE\_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, 2*N);
        elseif fptype == FPTYPE\_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign, 2*N);
        else
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFMaxNum

```
// BFMaxNum()
// =====

bits(N) BFMaxNum(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMaxNum(op1, op2, fpcr, fpexc);

// 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.

bits(N) BFMaxNum(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    constant boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    constant boolean type2_nan = 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('1', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity('1', N);

    constant boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMax(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFMin

```
// BFMin()
// =====
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return BFMin(op1, op2, fpcr, altfp, fpexc);

// BFMin()
// =====
// BFloat16 minimum.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return BFMin(op1, op2, fpcr, altfp, fpexc);

// 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.

bits(N) BFMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N == 16;
    FPCR\_Type fpcr = fpcr_in;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return BFZero(sign2, N);
    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);
        return (if type2 == FPTYPE\_Zero then BFZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, 2*N);
        elseif fptype == FPTYPE\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign, 2*N);
        else
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
            result = FPRoundBF(value, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFMinNum

```
// BFMinNum()
// =====

bits(N) BFMinNum(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMinNum(op1, op2, fpcr, fpexc);

// 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.

bits(N) BFMinNum(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant boolean isbfloat16 = TRUE;
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    bits(N) result;

    (type1,-,-) = FPUnpackBase(op1, fpcr, fpexc, isbfloat16);
    (type2,-,-) = FPUnpackBase(op2, fpcr, fpexc, isbfloat16);

    constant boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    constant boolean type2_nan = 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('0', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = BFInfinity('0', N);

    constant boolean altfmaxfmin = FALSE;    // Do not use alternate NaN handling
    result = BFMin(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFMul

```
// BFMul()
// =====
// Non-widening BFloat16 multiply used by SVE2 instructions.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMul(op1, op2, fpcr, fpexc);

// 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.

bits(N) BFMul(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType Infinity);
        inf2 = (type2 == FPType Infinity);
        zero1 = (type1 == FPType Zero);
        zero2 = (type2 == FPType Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = FPRoundBF(value1*value2, fpcr, rounding, fpexc, 2*N);

        if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```



```

// BFMulAdd()
// =====
// Non-widening BFloat16 fused multiply-add used by SVE2 instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

// 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.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) addend_s = addend : Zeros(N);
    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (typeA,signA,valueA) = FPUnpack(addend_s, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    inf1 = (type1 == FPTYPE\_Infinity);
    inf2 = (type2 == FPTYPE\_Infinity);
    zero1 = (type1 == FPTYPE\_Zero);
    zero2 = (type2 == FPTYPE\_Zero);

    (done,result) = FPProcessNaNs3(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(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE\_Infinity);
        zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 2*N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 2*N);

        // 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(signA, 2*N);

        // Otherwise calculate numerical result and round it.

```

```

else
    result_value = valueA + (value1 * value2);
    if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
        result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
        result = FPZero(result_sign, 2*N);
    else
        result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

    if !invalidop && fpexc then
        FPProcessDenorms3(typeA, type1, type2, 2*N, fpcr);

return result<2*N-1:N>;

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH

```

// BFMulAddH()
// =====
// Used by BFMLALB, BFMLALT, BFMLSLB and BFMLSLT instructions.

bits(32) BFMulAddH(bits(32) addend, bits(16) op1, bits(16) op2, FPCR\_Type fpcr_in)
    constant bits(32) value1 = op1 : Zeros(16);
    constant bits(32) value2 = op2 : Zeros(16);
    FPCR\_Type fpcr = fpcr_in;
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    // When using alternative floating-point behaviour, do not generate floating-point exceptions
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and
                                        // output to zero
    if altfp then fpcr.RMode = '00'; // Use RNE rounding mode
    return FPMulAdd(addend, value1, value2, fpcr, fpexc);

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAddH\_ZA

```

// BFMulAddH_ZA()
// =====
// Used by SME2 ZA-targeting BFMLAL and BFMLSL instructions.

bits(32) BFMulAddH_ZA(bits(32) addend, bits(16) op1, bits(16) op2, FPCR\_Type fpcr)
    constant bits(32) value1 = op1 : Zeros(16);
    constant bits(32) value2 = op2 : Zeros(16);
    return FPMulAdd\_ZA(addend, value1, value2, fpcr);

```

### Library pseudocode for shared/functions/float/bfloat/BFMulAdd\_ZA

```

// BFMulAdd_ZA()
// =====
// Non-widening BFloat16 fused multiply-add used by SME2 ZA-targeting instructions.

bits(N) BFMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    constant boolean fpexc = FALSE;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFMulAdd(addend, op1, op2, fpcr, fpexc);

```



## Library pseudocode for shared/functions/float/bfloat/BFMulH

```
// BFMulH()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(2*N) BFMulH(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N == 16;
    bits(2*N) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FType\_QNaN || type2 == FType\_QNaN then
        result = FPDefaultNaN(fpcr, 2*N);
    else
        inf1 = (type1 == FType\_Infinity);
        inf2 = (type2 == FType\_Infinity);
        zero1 = (type1 == FType\_Zero);
        zero2 = (type2 == FType\_Zero);
        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, 2*N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, 2*N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, 2*N);
        else
            result = BFRound(value1*value2, 2*N);

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

bits(N) BFNeg(bits(N) op)
    assert N == 16;
    constant boolean honor_alttp = TRUE;    // Honor alternate handling
    return BFNeg(op, honor_alttp);

// BFNeg()
// =====

bits(N) BFNeg(bits(N) op, boolean honor_alttp)
    assert N == 16;
    if honor_alttp && !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if FPCR.AH == '1' then
            constant boolean fpexc = FALSE;
            constant boolean isbfloat16 = TRUE;
            (fptype, -, -) = FPUnpackBase(op, FPCR, fpexc, isbfloat16);
            if fptype IN {FType\_SNaN, FType\_QNaN} then
                return op;    // When FPCR.AH=1, sign of NaN has no consequence
    return NOT(op<N-1>) : op<N-2:0>;
```

## 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.

bits(N) BFRound(real op, integer N)
    assert N == 32;
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant integer minimum_exp = -126;  constant integer E = 8;  constant integer F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1';  mantissa = -op;
    else
        sign = '0';  mantissa = op;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign, N);

    // 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).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F);  // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    // Round to Odd
    if error != 0.0 && int_mant<0> == '0' then
        int_mant = int_mant + 1;

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign, N);  // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<(N-2)-F:0> : int_mant<F-1:0>;

    return result;
```

## Library pseudocode for shared/functions/float/bfloat/BFScale

```
// BFScale()
// =====
// Scales BFloat16 operand by 2.0 to the power of the signed integer value.

bits(N) BFScale(bits(N) op, integer scale, FPCR\_Type fpcr)
    assert N == 16;
    bits(2*N) result;

    constant bits(2*N) op_s = op : Zeros(N);
    (fptype,sign,value) = FPUnpack(op_s, fpcr);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN(fptype, op_s, fpcr);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero(sign, 2*N);
    elsif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign, 2*N);
    else
        constant FPRounding rounding = FPRoundingMode(fpcr);
        constant boolean fpexc = TRUE;
        result = FPRoundBF(value * (2.0^scale), fpcr, rounding, fpexc, 2*N);
        if fpexc then FPProcessDenorm(fptype, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFSub

```
// BFSub()
// =====
// Non-widening BFloat16 subtraction used by SVE2 instructions.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return BFSub(op1, op2, fpcr, fpexc);

// 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.

bits(N) BFSub(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)
    assert N == 16;
    constant FPRounding rounding = FPRoundingMode(fpcr);
    boolean done;
    bits(2*N) result;

    constant bits(2*N) op1_s = op1 : Zeros(N);
    constant bits(2*N) op2_s = op2 : Zeros(N);
    (type1,sign1,value1) = FPUnpack(op1_s, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2_s, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1_s, op2_s, fpcr, fpexc);

    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, 2*N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', 2*N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', 2*N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, 2*N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, 2*N);
            else
                result = FPRoundBF(result_value, fpcr, rounding, fpexc, 2*N);

            if fpexc then FPProcessDenorms(type1, type2, 2*N, fpcr);

    return result<2*N-1:N>;
```

## Library pseudocode for shared/functions/float/bfloat/BFSub\_ZA

```
// BFSub_ZA()
// =====
// Non-widening BFloat16 subtraction used by SME2 ZA-targeting instructions.

bits(N) BFSub_ZA(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    constant boolean fpexc = FALSE;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.DN = '1'; // Generate default NaN values
    return BFSub(op1, op2, fpcr, fpexc);
```

## 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.)

(FPType, bit, real) BFUnpack(bits(N) fpval)
    assert N IN {16,32};

    bit sign;
    bits(8) exp;
    bits(23) frac;
    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>;

    FPType fptype;
    real value;
    if IsZero(exp) then
        fptype = FPType Zero; value = 0.0; // Fixed Flush to Zero
    elseif IsOnes(exp) then
        if IsZero(frac) then
            fptype = FPType Infinity; value = 2.0^1000000;
        else // no SNaN for BF16 arithmetic
            fptype = FPType QNaN; value = 0.0;
    else
        fptype = FPType Nonzero;
        value = 2.0^(UInt(exp)-127) * (1.0 + Real(UInt(frac)) * 2.0^-23);

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

## Library pseudocode for shared/functions/float/bfloat/BFZero

```
// BFZero()
// =====

bits(N) BFZero(bit sign, integer N)
    assert N == 16;
    constant integer E = 8;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

## Library pseudocode for shared/functions/float/bfloat/FAdd\_BF16

```
// FAdd_BF16()
// =====
// Single-precision add following BFloat16 computation behaviors.

bits(N) FAdd_BF16(bits(N) op1, bits(N) op2, FPCR Type fpcr)
    assert N == 32;
    bits(N) result;

    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPTYPE\_QNaN || type2 == FPTYPE\_QNaN then
        result = FPDefaultNaN(fpcr, N);
    else
        inf1 = (type1 == FPTYPE\_Infinity);
        inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);
        zero2 = (type2 == FPTYPE\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
        elseif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elseif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elseif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0', N);    // Positive sign when Round to Odd
            else
                result = BFRound(result_value, N);

    return result;
```

## 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.

bits(16) FPConvertBF(bits(32) op, FPCR\_Type fpcr_in, FPRounding rounding_in)
    constant integer halfsize = 16;
    FPCR\_Type fpcr = fpcr_in;
    FPRounding rounding = rounding_in;
    bits(32) result; // BF16 value in top 16 bits
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    if altfp then rounding = FPRounding\_TIEEVEN; // Use RNE rounding mode

    // Unpack floating-point operand, with always flush-to-zero if fpcr.AH == '1'.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        if fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, 32);
        else
            result = FPConvertNaN(op, 32);
        if fptype == FPTYPE\_SNaN then
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign, 32);
    elseif fptype == FPTYPE\_Zero then
        result = FPZero(sign, 32);
    else
        result = FPRoundBF(value, fpcr, rounding, fpexc, 32);

    // Returns correctly rounded BF16 value from top 16 bits
    return result<(2*halfsize)-1:halfsize>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCR\_Type fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

## 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.

bits(N) FPRoundBF(real op, FPCR\_Type fpcr, FPRounding rounding, boolean fpexc, integer N)
    assert N == 32;
    constant boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```

## 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.

bits(N) FixedToFP(bits(M) op, integer fbits, boolean unsigned, FPCR\_Type fpcr,
FPRounding rounding, integer N)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(N) result;
    assert fbits >= 0;
    assert rounding != FPRounding\_ODD;

    // Correct signed-ness
    int_operand = Int(op, unsigned);

    // Scale by fractional bits and generate a real value
    real_operand = Real(int_operand) / 2.0^fbits;

    if real_operand == 0.0 then
        result = FPZero('0', N);
    else
        result = FPRound(real_operand, fpcr, rounding, N);

    return result;
```

## Library pseudocode for shared/functions/float/fp8float/BFConvertFP8

```
// BFConvertFP8()
// =====
// Converts a BFloat16 OP to FP8 value.

bits(8) BFConvertFP8(bits(16) op_in, FPCR\_Type fpcr, FPMR\_Type fpmr)
    constant bits(32) op = op_in : Zeros(16);
    return FPConvertFP8(op, fpcr, fpmr, 8);
```

## Library pseudocode for shared/functions/float/fp8float/FP8Bits

```
// FP8Bits()
// =====
// Returns the minimum exponent, numbers of exponent and fraction bits.

FP8BitsType FP8Bits(FP8Type fp8type)
    integer minimum_exp;
    integer F;
    if fp8type == FP8Type\_OFP8\_E4M3 then
        minimum_exp = -6; F = 3;
    else // fp8type == FP8Type\_OFP8\_E5M2
        minimum_exp = -14; F = 2;

    return (F, minimum_exp);
```

## Library pseudocode for shared/functions/float/fp8float/FP8ConvertBF

```
// FP8ConvertBF()
// =====
// Converts an FP8 operand to BFloat16 value.

bits(2*N) FP8ConvertBF(bits(N) op, boolean issrc2, FPCR\_Type fpcr, FPMR\_Type fpmr)
    assert N == 8;
    constant boolean isbfloat16 = TRUE;
    constant bits(4*N) result = FP8ConvertFP(op, issrc2, fpcr, fpmr, isbfloat16, 4*N);
    return result<2*N+:2*N>;
```



## Library pseudocode for shared/functions/float/fp8float/FP8ConvertFP

```
// FP8ConvertFP()
// =====
// Converts an FP8 operand to half-precision value.

bits(2*N) FP8ConvertFP(bits(N) op, boolean issrc2, FPCR\_Type fpcr, FPMR\_Type fpmr)
    assert N == 8;
    constant boolean isbfloat16 = FALSE;
    return FP8ConvertFP(op, issrc2, fpcr, fpmr, isbfloat16, 2*N);

// 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.

bits(M) FP8ConvertFP(bits(N) op, boolean issrc2, FPCR\_Type fpcr_in, FPMR\_Type fpmr,
    boolean isbfloat16, integer M)
    assert N == 8 && M IN {16,32};
    bits(M) result;

    constant boolean fpexc = TRUE;
    FPCR\_Type fpcr = fpcr_in;
    // Do not flush denormal inputs and outputs to zero.
    // Do not support alternative half-precision format.
    fpcr.<FIZ,FZ,FZ16,AHP> = '0000';
    rounding = FPRounding\_TIEEVEN;
    constant FP8Type fp8type = (if issrc2 then FP8DecodeType(fpmr.F8S2)
        else FP8DecodeType(fpmr.F8S1));

    (fptype,sign,value) = FP8Unpack(op, fp8type);

    if fptype == FPType\_SNaN || fptype == FPType\_QNaN then
        result = FPDefaultNaN(fpcr, M);
        if fptype == FPType\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPType\_Infinity then
        result = FPInfinity(sign, M);
    elseif fptype == FPType\_Zero then
        result = FPZero(sign, M);
    else
        integer dscale;
        if issrc2 then
            dscale = (if M == 16 then UInt(fpmr.LSCALE2<3:0>)
                else UInt(fpmr.LSCALE2<5:0>));
        else
            dscale = (if M == 16 then UInt(fpmr.LSCALE<3:0>)
                else UInt(fpmr.LSCALE<5:0>));
        constant real result_value = value * (2.0^-dscale);
        result = FPRoundBase(result_value, fpcr, rounding, isbfloat16, fpexc, M);

    return result;
```

## Library pseudocode for shared/functions/float/fp8float/FP8DecodeType

```
// FP8DecodeType()
// =====
// Decode the FP8 format encoded in F8S1, F8S2 or F8D field in FPMR

FP8Type FP8DecodeType(bits(3) f8format)
    case f8format of
        when '000' return FP8Type\_OFP8\_E5M2;
        when '001' return FP8Type\_OFP8\_E4M3;
        otherwise return FP8Type\_UNSUPPORTED;
```

## Library pseudocode for shared/functions/float/fp8float/FP8DefaultNaN

```
// FP8DefaultNaN()
// =====

bits(N) FP8DefaultNaN(FP8Type fp8type, FPCR\_Type fpcr, integer N)
  assert N == 8;
  assert fp8type IN {FP8Type\_OFP8\_E5M2, FP8Type\_OFP8\_E4M3};
  constant bit sign = if IsFeatureImplemented(FEAT_AFP) then fpcr.AH else '0';
  constant integer E = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
  constant integer F = N - (E + 1);
  bits(E) exp;
  bits(F) frac;

  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);

  return sign : exp : frac;
```



```

// FP8DotAddFP()
// =====

bits(M) FP8DotAddFP(bits(M) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr, FPMR\_Type fpmr)
    constant integer E = N DIV 8;
    return FP8DotAddFP(addend, op1, op2, E, fpcr, fpmr);

// 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.

bits(M) FP8DotAddFP(bits(M) addend, bits(N) op1, bits(N) op2, integer E,
    FPCR\_Type fpcr_in, FPMR\_Type fpmr)
    assert M IN {16,32};
    assert N IN {2*M, M, M DIV 2, M DIV 4};
    FPCR\_Type fpcr = fpcr_in;
    bits(M) result;

    fpcr.<FIZ,FZ,FZ16> = '000';          // Do not flush denormal inputs and outputs to zero
    fpcr.DN = '1';
    rounding = FPRounding\_TIEEVEN;

    constant FP8Type fp8type1 = FP8DecodeType(fpmr.F8S1);
    constant FP8Type fp8type2 = FP8DecodeType(fpmr.F8S2);

    array[0..(E-1)] of FPTType type1;
    array[0..(E-1)] of FPTType type2;
    array[0..(E-1)] of bit sign1;
    array[0..(E-1)] of bit sign2;
    array[0..(E-1)] of real value1;
    array[0..(E-1)] of real value2;
    array[0..(E-1)] of boolean inf1;
    array[0..(E-1)] of boolean inf2;
    array[0..(E-1)] of boolean zero1;
    array[0..(E-1)] of boolean zero2;

    constant boolean fpexc = FALSE;
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    infA = (typeA == FPTType\_Infinity);    zeroA = (typeA == FPTType\_Zero);
    boolean any_nan = typeA IN {FPTType\_SNaN, FPTType\_QNaN};
    for i = 0 to E-1
        (type1[i], sign1[i], value1[i]) = FP8Unpack(Elem[op1, i, N DIV E], fp8type1);
        (type2[i], sign2[i], value2[i]) = FP8Unpack(Elem[op2, i, N DIV E], fp8type2);
        inf1[i] = (type1[i] == FPTType\_Infinity); zero1[i] = (type1[i] == FPTType\_Zero);
        inf2[i] = (type2[i] == FPTType\_Infinity); zero2[i] = (type2[i] == FPTType\_Zero);
        any_nan = (any_nan || type1[i] IN {FPTType\_SNaN, FPTType\_QNaN} ||
            type2[i] IN {FPTType\_SNaN, FPTType\_QNaN});

    if any_nan then
        result = FPDefaultNaN(fpcr, M);
    else
        // Determine sign and type products will have if it does not cause an Invalid
        // Operation.
        array [0..(E-1)] of bit signP;
        array [0..(E-1)] of boolean infP;
        array [0..(E-1)] of boolean zeroP;
        for i = 0 to E-1
            signP[i] = sign1[i] EOR sign2[i];
            infP[i] = inf1[i] || inf2[i];
            zeroP[i] = zero1[i] || zero2[i];

        // Detect non-numeric results of dot product and accumulate
        boolean posInfR = (infA && signA == '0');
        boolean negInfR = (infA && signA == '1');

```

```

boolean zeroR = zeroA;
boolean invalidop = FALSE;
for i = 0 to E-1
    // 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
        invalidop = invalidop || (infP[i] && infP[j] && (signP[i] != signP[j]));

if invalidop then
    result = FPDefaultNaN(fpcr, M);

// Other cases involving infinities produce an infinity of the same sign.
elseif posInfR then
    result = FPInfinity('0', M);
elseif negInfR then
    result = FPInfinity('1', M);

// 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(signA, M);

// Otherwise calculate numerical value and round it.
else
    // Apply scaling to sum-of-product
    constant integer dscale = if M == 32 then UInt(fpmr.LSCALE) else UInt(fpmr.LSCALE<3:0>);

    real dp_value = value1[0] * value2[0];
    for i = 1 to E-1
        dp_value = dp_value + value1[i] * value2[i];

    constant real result_value = 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('0', M);
    else
        constant boolean satoflo = (fpmr.OSM == '1');
        result = FPRound\_FP8(result_value, fpcr, rounding, satoflo, M);

return result;

```

## Library pseudocode for shared/functions/float/fp8float/FP8Infinity

```

// FP8Infinity()
// =====

bits(N) FP8Infinity(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type\_OF8\_E5M2, FP8Type\_OF8\_E4M3};
    constant integer E = if fp8type == FP8Type\_OF8\_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    bits(E) exp;
    bits(F) frac;

    case fp8type of
        when FP8Type\_OF8\_E4M3
            exp = Ones(E);
            frac = Ones(F);
        when FP8Type\_OF8\_E5M2
            exp = Ones(E);
            frac = Zeros(F);

    return sign : exp : frac;

```

## 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])

bits(N) FP8MatMulAddFP(bits(N) addend, bits(N) op1, bits(N) op2, integer E, FPCR\_Type fpcr,
                        FPMR\_Type fpmr)
    assert N IN {64, 128};
    assert N == E*16;
    constant integer M = N DIV 4;
    bits(N) result;

    for i = 0 to 1
        for j = 0 to 1
            constant bits(2*M) elt1 = Elem[op1, i, 2*M];
            constant bits(2*M) elt2 = Elem[op2, j, 2*M];
            constant bits(M) sum = Elem[addend, 2*i + j, M];
            Elem[result, 2*i + j, M] = FP8DotAddFP(sum, elt1, elt2, E, fpcr, fpmr);

    return result;
```

## Library pseudocode for shared/functions/float/fp8float/FP8MaxNormal

```
// FP8MaxNormal()
// =====

bits(N) FP8MaxNormal(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type\_OFP8\_E5M2, FP8Type\_OFP8\_E4M3};
    constant integer E = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    bits(E) exp;
    bits(F) frac;

    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);

    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fp8float/FP8MulAddFP

```
// FP8MulAddFP()
// =====

bits(M) FP8MulAddFP(bits(M) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr,
                    FPMR\_Type fpmr)
    assert N == 8 && M IN {16, 32};
    constant integer E = 1;
    return FP8DotAddFP(addend, op1, op2, E, fpcr, fpmr);
```



```

// FP8Round()
// =====
// Used by FP8 downconvert instructions which observe FPMR.OSC
// to convert a real number OP into an FP8 value.

bits(N) FP8Round(real op, FP8Type fp8type, FPCR\_Type fpcr, FPMR\_Type fpmr, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type\_OF8\_E5M2, FP8Type\_OF8\_E4M3};
    assert op != 0.0;
    bits(N) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant (F, minimum_exp) = FP8Bits(fp8type);
    constant E = (N - F) - 1;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;

    (mantissa, exponent) = NormalizeReal(mantissa);
    // When TRUE, detection of underflow occurs after rounding.
    altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = 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).
    biased_exp = Max((exponent - minimum_exp) + 1, 0);
    if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

    // Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
    int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
    error = mantissa * 2.0^F - Real(int_mant);

    constant boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());

    boolean round_up_unconstrained;
    boolean round_up;

    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;

        // Follow alternate floating-point behavior of underflow after rounding
        if (biased_exp_unconstrained < 1 && int_mant_unconstrained != 0 &&
            (error != 0.0 || trapped_UF)) then
            FPPProcessException(FPExc\_Underflow, fpcr);
    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
            FPPProcessException(FPExc\_Underflow, fpcr);

        // Round to Nearest Even
        round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));

```



```

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then          // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then      // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;

// Deal with overflow and generate result.
boolean overflow;
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;

if overflow then
    result = (if fpmr.OSC == '0' then FP8Infinity(fp8type, sign, N)
              else FP8MaxNormal(fp8type, sign, N));
    // Flag Overflow exception regardless of FPMR.OSC
    FPProcessException(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>;

// Deal with Inexact exception.
if error != 0.0 then
    FPProcessException(FPExc\_Inexact, fpcr);

return result;

```

### Library pseudocode for shared/functions/float/fp8float/FP8Type

```

// FP8Type
// =====

enumeration FP8Type {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.

(FPType, bit, real) FP8Unpack(bits(N) fpval, FP8Type fp8type)
    assert N == 8;
    constant integer E = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);

    constant bit sign = fpval<N-1>;
    constant bits(E) exp = fpval<(E+F)-1:F>;
    constant bits(F) frac = fpval<F-1:0>;

    real value;
    FPType fptype;

    if fp8type == FP8Type\_OFP8\_E4M3 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType\_Zero; value = 0.0;
            else
                fptype = FPType\_Denormal; value = 2.0^-6 * (Real(UInt(frac)) * 2.0^-3);
        elsif IsOnes(exp) && IsOnes(frac) then
            fptype = FPType\_SNaN;
            value = 0.0;
        else
            fptype = FPType\_Nonzero;
            value = 2.0^(UInt(exp)-7) * (1.0 + Real(UInt(frac)) * 2.0^-3);

    elsif fp8type == FP8Type\_OFP8\_E5M2 then
        if IsZero(exp) then
            if IsZero(frac) then
                fptype = FPType\_Zero; value = 0.0;
            else
                fptype = FPType\_Denormal; value = 2.0^-14 * (Real(UInt(frac)) * 2.0^-2);
        elsif IsOnes(exp) then
            if IsZero(frac) then
                fptype = FPType\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac<1> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            fptype = FPType\_Nonzero;
            value = 2.0^(UInt(exp)-15) * (1.0 + Real(UInt(frac)) * 2.0^-2);

    else // fp8type == FP8Type_UNSUPPORTED
        fptype = FPType\_SNaN;
        value = 0.0;

    if sign == '1' then value = -value;

    return (fptype, sign, value);
```

## Library pseudocode for shared/functions/float/fp8float/FP8Zero

```
// FP8Zero()
// =====

bits(N) FP8Zero(FP8Type fp8type, bit sign, integer N)
    assert N == 8;
    assert fp8type IN {FP8Type\_OFP8\_E5M2, FP8Type\_OFP8\_E4M3};
    constant integer E = if fp8type == FP8Type\_OFP8\_E4M3 then 4 else 5;
    constant integer F = N - (E + 1);
    return sign : Zeros(E) : Zeros(F);
```

## 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.

bits(M) FPConvertFP8(bits(N) op, FPCR\_Type fpcr_in, FPMR\_Type fpmr, integer M)
    assert N IN {16,32} && M == 8;
    bits(M) result;

    constant boolean fpexc = TRUE;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.<FIZ,FZ,FZ16> = '000'; // Do not flush denormal inputs and outputs to zero
    constant FP8Type fp8type = FP8DecodeType(fpmr.F8D);

    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fp8type == FP8Type\_UNSUPPORTED then
        result = Ones(M);
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FP8DefaultNaN(fp8type, fpcr, M); // Always generate Default NaN as result
        if fptype == FPTYPE\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTYPE\_Infinity then
        result = (if fpmr.OSC == '0' then FP8Infinity(fp8type, sign, M)
            else FP8MaxNormal(fp8type, sign, M));
    elsif fptype == FPTYPE\_Zero then
        result = FP8Zero(fp8type, sign, M);
    else
        constant integer scale = if N == 16 then SInt(fpmr.NSCALE<4:0>) else SInt(fpmr.NSCALE);
        constant real result_value = value * (2.0^scale);
        result = FP8Round(result_value, fp8type, fpcr, fpmr, M);

    return result;
```

## Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTYPE\_SNaN, FPTYPE\_QNaN} then
                return op; // When fpcr.AH=1, sign of NaN has no consequence
    return '0' : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fpabsmax/FPAbsMax

```
// FPAbsMax()
// =====
// Compare absolute value of two operands and return the larger absolute
// value without rounding.

bits(N) FPAbsMax(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.<AH,FIZ,FZ,FZ16> = '0000';

    op1 = '0':op1_in<N-2:0>;
    op2 = '0':op2_in<N-2:0>;
    (type1,-,value1) = FPUnpack(op1, fpcr);
    (type2,-,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(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;

    return result;
```

## Library pseudocode for shared/functions/float/fpabsmin/FPAbsMin

```
// FPAbsMin()
// =====
// Compare absolute value of two operands and return the smaller absolute
// value without rounding.

bits(N) FPAbsMin(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr_in)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR\_Type fpcr = fpcr_in;
    fpcr.<AH,FIZ,FZ,FZ16> = '0000';

    op1 = '0':op1_in<N-2:0>;
    op2 = '0':op2_in<N-2:0>;
    (type1,-,value1) = FPUnpack(op1, fpcr);
    (type2,-,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(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;

    return result;
```

## Library pseudocode for shared/functions/float/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity); inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero); zero2 = (type2 == FPTYPE\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1, N);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpadd/FPAdd\_ZA

```
// FPAdd_ZA()
// =====
// Calculates op1+op2 for SME2 ZA-targeting instructions.

bits(N) FPAdd_ZA(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    return FPAdd(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpbits/FPBits

```
// FPBits()
// =====
// Returns the minimum exponent, numbers of exponent and fraction bits.

FPBitsType FPBits(integer N, boolean isbfloat16)
  FPFracBits F;
  integer minimum_exp;
  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;

  return (F, minimum_exp);
```

## Library pseudocode for shared/functions/float/fpbits/FPBitsType

```
// FPBitsType
// =====

type FPBitsType = (FPFracBits, integer);
```

## Library pseudocode for shared/functions/float/fpbits/FPFracBits

```
// FPFracBits
// =====

type FPFracBits = integer;
```

## Library pseudocode for shared/functions/float/fpcompare/FPCompare

```
// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCR\_Type fpcr)
  assert N IN {16,32,64};
  (type1,sign1,value1) = FPUnpack(op1, fpcr);
  (type2,sign2,value2) = FPUnpack(op2, fpcr);

  bits(4) result;
  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);
  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';

    FPProcessDenorms(type1, type2, N, fpcr);
  return result;
```

## Library pseudocode for shared/functions/float/fpcompareeq/FPCmpareEQ

```
// FPCmpareEQ()
// =====

boolean FPCmpareEQ(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTypE\_SNaN, FPTypE\_QNaN} || type2 IN {FPTypE\_SNaN, FPTypE\_QNaN} then
        result = FALSE;
        if type1 == FPTypE\_SNaN || type2 == FPTypE\_SNaN then
            FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 == value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpcomparege/FPCmpareGE

```
// FPCmpareGE()
// =====

boolean FPCmpareGE(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTypE\_SNaN, FPTypE\_QNaN} || type2 IN {FPTypE\_SNaN, FPTypE\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 >= value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpcomparegt/FPCmpareGT

```
// FPCmpareGT()
// =====

boolean FPCmpareGT(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTypE\_SNaN, FPTypE\_QNaN} || type2 IN {FPTypE\_SNaN, FPTypE\_QNaN} then
        result = FALSE;
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUntpack()
        result = (value1 > value2);
        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## 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.

bits(M) FPConvert(bits(N) op, FPCR\_Type fpcr, FPRounding rounding, integer M)

    assert M IN {16,32,64};
    assert N IN {16,32,64};
    bits(M) result;

    // Unpack floating-point operand optionally with flush-to-zero.
    (fptype,sign,value) = FPUnpackCV(op, fpcr);

    alt_hp = (M == 16) && (fpcr.AHP == '1');

    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        if alt_hp then
            result = FPZero(sign, M);
        elseif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr, M);
        else
            result = FPConvertNaN(op, M);
            if fptype == FPTYPE\_SNaN || alt_hp then
                FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTYPE\_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else
            result = FPInfinity(sign, M);
    elseif fptype == FPTYPE\_Zero then
        result = FPZero(sign, M);
    else
        result = FPRoundCV(value, fpcr, rounding, M);
        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCR\_Type fpcr, integer M)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr), M);
```



## Library pseudocode for shared/functions/float/fpconvertnan/FPConvertNaN

```
// FPConvertNaN()
// =====
// Converts a NaN of one floating-point type to another

bits(M) FPConvertNaN(bits(N) op, integer M)
    assert N IN {16,32,64};
    assert M IN {16,32,64};
    bits(M) result;
    bits(51) frac;

    sign = 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);

    // 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>;

    return result;
```

## Library pseudocode for shared/functions/float/fpdecoderm/FPDecodeRM

```
// FPDecodeRM()
// =====

// Decode most common AArch32 floating-point rounding encoding.

FPRounding FPDecodeRM(bits(2) rm)
    FPRounding result;
    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

    return result;
```

## Library pseudocode for shared/functions/float/fpdecoderounding/FPDecodeRounding

```
// FPDecodeRounding()
// =====

// Decode floating-point rounding mode and common AArch64 encoding.

FPRounding FPDecodeRounding(bits(2) rmode)
    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
```

## Library pseudocode for shared/functions/float/fpdefaultnan/FPDefaultNaN

```
// FPDefaultNaN()
// =====

bits(N) FPDefaultNaN(FPCR\_Type fpcr, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    constant bit sign = if IsFeatureImplemented(FEAT_AFP) && !UsingAArch32\(\) then fpcr.AH else '0';

    constant bits(E) exp = Ones(E);
    constant bits(F) frac = '1':Zeros(F-1);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpdiv/FPDiv

```
// FPDiv()
// =====

bits(N) FPDiv(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);

    if !done then
        inf1 = type1 == FPTYPE\_Infinity;
        inf2 = type2 == FPTYPE\_Infinity;
        zero1 = type1 == FPTYPE\_Zero;
        zero2 = type2 == FPTYPE\_Zero;

        if (inf1 && inf2) || (zero1 && zero2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPIInfinity(sign1 EOR sign2, N);
            if !inf1 then FPProcessException(FPExc\_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1/value2, fpcr, N);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```



```

// 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.

bits(32) FPDot(bits(16) op1_a, bits(16) op1_b, bits(16) op2_a,
               bits(16) op2_b, FPCR\_Type fpcr, boolean isbfloat16)
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);

bits(32) FPDot(bits(16) op1_a, bits(16) op1_b, bits(16) op2_a,
               bits(16) op2_b, FPCR\_Type fpcr_in, boolean isbfloat16, boolean fpexc)
    FPCR\_Type fpcr = fpcr_in;
    bits(32) result;
    boolean done;
    fpcr.AHP = '0'; // Ignore alternative half-precision option
    rounding = FPRoundingMode(fpcr);

    (type1_a, sign1_a, value1_a) = FPUnpackBase(op1_a, fpcr, fpexc, isbfloat16);
    (type1_b, sign1_b, value1_b) = FPUnpackBase(op1_b, fpcr, fpexc, isbfloat16);
    (type2_a, sign2_a, value2_a) = FPUnpackBase(op2_a, fpcr, fpexc, isbfloat16);
    (type2_b, sign2_b, value2_b) = FPUnpackBase(op2_b, fpcr, fpexc, isbfloat16);

    inf1_a = (type1_a == FPTYPE\_Infinity); zero1_a = (type1_a == FPTYPE\_Zero);
    inf1_b = (type1_b == FPTYPE\_Infinity); zero1_b = (type1_b == FPTYPE\_Zero);
    inf2_a = (type2_a == FPTYPE\_Infinity); zero2_a = (type2_a == FPTYPE\_Zero);
    inf2_b = (type2_b == FPTYPE\_Infinity); 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.
        signPa = sign1_a EOR sign2_a;
        signPb = sign1_b EOR sign2_b;
        infPa = inf1_a || inf2_a;
        infPb = inf1_b || inf2_b;
        zeroPa = zero1_a || zero2_a;
        zeroPb = zero1_b || zero2_b;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        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(fpcr, 32);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infPa && signPa == '0') || (infPb && signPb == '0') then
            result = FPInfinity('0', 32);
        elsif (infPa && signPa == '1') || (infPb && signPb == '1') then
            result = FPInfinity('1', 32);

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroPa && zeroPb && signPa == signPb then
            result = FPZero(signPa, 32);

        // Otherwise calculate fused sum of products and round it.
        else
            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
                result_sign = FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);

```

```

        else
            result = FPRound(result_value, fpcr, rounding, fpexc, 32);

    return result;

```

### Library pseudocode for shared/functions/float/fpdot/FPDotAdd

```

// FPDotAdd()
// =====
// Half-precision 2-way dot-product and add to single-precision.

bits(32) FPDotAdd(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                  bits(16) op2_a, bits(16) op2_b, FPCR\_Type fpcr)
    bits(32) prod;
    constant boolean isbfloat16 = FALSE;
    constant boolean fpexc = TRUE;      // Generate floating-point exceptions
    prod = FPDot(op1_a, op1_b, op2_a, op2_b, fpcr, isbfloat16, fpexc);
    result = FPAdd(addend, prod, fpcr, fpexc);

    return result;

```

### 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.

bits(32) FPDotAdd_ZA(bits(32) addend, bits(16) op1_a, bits(16) op1_b,
                    bits(16) op2_a, bits(16) op2_b, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    bits(32) prod;
    constant boolean isbfloat16 = FALSE;
    constant boolean fpexc = 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(addend, prod, fpcr, fpexc);

    return result;

```

### Library pseudocode for shared/functions/float/fpexc/FPExc

```

// FPExc
// =====

enumeration FPExc      {FPExc_InvalidOp, FPExc_DivideByZero, FPExc_Overflow,
                       FPExc_Underflow, FPExc_Inexact, FPExc_InputDenorm};

```

### Library pseudocode for shared/functions/float/fpinfinity/FPInfinity

```

// FPinfinity()
// =====

bits(N) FPinfinity(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    constant bits(E) exp  = Ones(E);
    constant bits(F) frac = Zeros(F);
    return sign : exp : frac;

```

## 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])

bits(N) FPMatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, ESize esize, FPCR\_Type fpcr)
    assert N == esize * 2 * 2;
    bits(N) result;
    bits(esize) prod0, prod1, sum;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, esize];
            prod0 = FPMul(Elem[op1, 2*i + 0, esize],
                          Elem[op2, 2*j + 0, esize], fpcr);
            prod1 = FPMul(Elem[op1, 2*i + 1, esize],
                          Elem[op2, 2*j + 1, esize], fpcr);
            sum = FPAdd(sum, FPAdd(prod0, prod1, fpcr), fpcr);
            Elem[result, 2*i + j, esize] = sum;

    return result;
```

## 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])

bits(N) FPMatMulAddH(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N == 128;
    constant integer M = 32;
    bits(N) result;

    constant boolean isbfloat16 = FALSE;
    for i = 0 to 1
        for j = 0 to 1
            bits(M) sum = Elem[addend, 2*i + j, M];
            array[0..1] of bits(M) prod;
            for k = 0 to 1
                constant bits(M DIV 2) elt1_a = Elem[op1, 4*i + 2*k + 0, M DIV 2];
                constant bits(M DIV 2) elt1_b = Elem[op1, 4*i + 2*k + 1, M DIV 2];
                constant bits(M DIV 2) elt2_a = Elem[op2, 4*j + 2*k + 0, M DIV 2];
                constant bits(M DIV 2) elt2_b = Elem[op2, 4*j + 2*k + 1, M DIV 2];
                prod[k] = FPDot(elt1_a, elt1_b, elt2_a, elt2_b, fpcr, isbfloat16);
            sum = FPAdd(sum, FPAdd(prod[0], prod[1], fpcr), fpcr);
            Elem[result, 2*i + j, M] = sum;

    return result;
```

## Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return FPMax(op1, op2, fpcr, altfp, fpexc);

// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return FPMax(op1, op2, fpcr, altfp, fpexc);

// 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.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR\_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif 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);
        return (if type2 == FPTYPE\_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPTYPE\_Zero then
            sign = sign1 AND sign2; // Use most positive sign
            result = FPZero(sign, N);
        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';

            result = FPRound(value, fpcr, rounding, fpexc, N);
            if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = Ones(E-1):'0';
    frac = Ones(F);
    return sign : exp : frac;
```

## Library pseudocode for shared/functions/float/fpmaxnum/FPMaxNum

```
// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return FPMaxNum(op1, op2, fpcr, fpexc);

// FPMaxNum()
// =====

bits(N) FPMaxNum(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr, fpexc);
    (type2,-,-) = FPUnpack(op2, fpcr, fpexc);

    constant boolean type1_nan = type1 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    constant boolean type2_nan = type2 IN {FPTYPE\_QNaN, FPTYPE\_SNaN};
    constant boolean altfp = 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('1', N);
        elsif type1 != FPTYPE\_QNaN && type2 == FPTYPE\_QNaN then
            op2 = FPInfinity('1', N);

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMAX(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

## 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.

boolean IsMerging(FPCR\_Type fpcr)
    constant bit nep = (if IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
        !IsFullA64Enabled() then '0' else fpcr.NEP);
    return IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && nep == '1';
```



## Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32\(\) && fpcr.AH == '1';
    constant boolean fpexc = TRUE;
    return FPMin(op1, op2, fpcr, altfp, fpexc);

// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean altfp)
    constant boolean fpexc = TRUE;
    return FPMin(op1, op2, fpcr, altfp, fpexc);

// 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.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in, boolean altfp, boolean fpexc)
    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    FPCR\_Type fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    if altfp && type1 == FPTYPE\_Zero && type2 == FPTYPE\_Zero && sign1 != sign2 then
        // Alternate handling of zeros with differing sign
        return FPZero(sign2, N);
    elsif 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);
        return (if type2 == FPTYPE\_Zero then FPZero(sign2, N) else op2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        FPTYPE fptype;
        bit sign;
        real value;
        FPRounding rounding;
        if value1 < value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FPTYPE\_Infinity then
            result = FPInfinity(sign, N);
        elsif fptype == FPTYPE\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign, N);
        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';

            result = FPRound(value, fpcr, rounding, fpexc, N);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;
    return FPMinNum(op1, op2, fpcr, fpexc);

// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCR\_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr, fpexc);
    (type2,-,-) = FPUnpack(op2, fpcr, fpexc);

    constant boolean type1_nan = type1 IN {FPType\_QNaN, FPType\_SNaN};
    constant boolean type2_nan = type2 IN {FPType\_QNaN, FPType\_SNaN};
    constant boolean altfp = 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('0', N);
        elsif type1 != FPType\_QNaN && type2 == FPType\_QNaN then
            op2 = FPInfinity('0', N);

    altfmaxfmin = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin, fpexc);

    return result;
```

## Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr, N);
            FPProcessException(FPExc_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```



```

// FPMulAdd()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

// 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.

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2,
    FPCR\_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};

    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    rounding = FPRoundingMode(fpcr);
    inf1 = (type1 == FPTYPE\_Infinity); zero1 = (type1 == FPTYPE\_Zero);
    inf2 = (type2 == FPTYPE\_Infinity); zero2 = (type2 == FPTYPE\_Zero);

    (done,result) = FPProcessNaNs3(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(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTYPE\_Infinity); zeroA = (typeA == FPTYPE\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero
        // by infinity and additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', N);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', N);

        // 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(signA, N);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

    if !invalidop && fpexc then

```

```

        FPProcessDenorms3(typeA, type1, type2, N, fpcr);
    return result;

```

## Library pseudocode for shared/functions/float/fpmuladd/FPMulAdd\_ZA

```

// FPMulAdd_ZA()
// =====
// Calculates addend + op1*op2 with a single rounding for SME ZA-targeting
// instructions.

bits(N) FPMulAdd_ZA(bits(N) addend, bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);

```



```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(32) FPMulAddH(bits(32) addend, bits(16) op1, bits(16) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE;          // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(32) FPMulAddH(bits(32) addend, bits(16) op1, bits(16) op2,
    FPCR\_Type fpcr, boolean fpexc)

    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUnpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPType\_Infinity); zero1 = (type1 == FPType\_Zero);
    inf2 = (type2 == FPType\_Infinity); zero2 = (type2 == FPType\_Zero);

    (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(fpcr, 32);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPType\_Infinity); zeroA = (typeA == FPType\_Zero);

        // Determine sign and type product will have if it does not cause an
        // Invalid Operation.
        signP = sign1 EOR sign2;
        infP = inf1 || inf2;
        zeroP = zero1 || zero2;

        // Non SNaN-generated Invalid Operation cases are multiplies of zero by infinity and
        // additions of opposite-signed infinities.
        invalidop = (inf1 && zero2) || (zero1 && inf2) || (infA && infP && signA != signP);

        if invalidop then
            result = FPDefaultNaN(fpcr, 32);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

        // Other cases involving infinities produce an infinity of the same sign.
        elsif (infA && signA == '0') || (infP && signP == '0') then
            result = FPInfinity('0', 32);
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1', 32);

        // 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(signA, 32);

        // Otherwise calculate numerical result and round it.
        else
            result_value = valueA + (value1 * value2);
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, 32);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, 32);

        if !invalidop && fpexc then
            FPProcessDenorm(typeA, 32, fpcr);

    return result;

```

## Library pseudocode for shared/functions/float/fpmuladdh/FPMulAddH\_ZA

```
// FPMulAddH_ZA()
// =====
// Calculates addend + op1*op2 for SME2 ZA-targeting instructions.

bits(32) FPMulAddH_ZA(bits(32) addend, bits(16) op1, bits(16) op2, FPCR\_Type fpcr_in)
FPCR\_Type fpcr = fpcr_in;
constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
fpcr.DN = '1'; // Generate default NaN values
return FPMulAddH(addend, op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpmuladdh/FPProcessNaNs3H

```
// FPProcessNaNs3H()
// =====

(boolean, bits(32)) FPProcessNaNs3H(FType type1, FType type2, FType type3,
                                     bits(32) op1, bits(16) op2, bits(16) op3,
                                     FPCR\_Type fpcr, boolean fpexc)

bits(32) result;
FType type_nan;
// When TRUE, use alternative NaN propagation rules.
constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
constant boolean op1_nan = type1 IN {FType\_SNaN, FType\_QNaN};
constant boolean op2_nan = type2 IN {FType\_SNaN, FType\_QNaN};
constant boolean op3_nan = type3 IN {FType\_SNaN, FType\_QNaN};
if altfp then
    if (type1 == FType\_SNaN || type2 == FType\_SNaN || type3 == FType\_SNaN) then
        type_nan = FType\_SNaN;
    else
        type_nan = FType\_QNaN;

boolean done;
if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), 32);
elseif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op2, fpcr, fpexc), 32);
elseif altfp && op3_nan && op1_nan then // <m> register NaN selected
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type_nan, op3, fpcr, fpexc), 32);
elseif type1 == FType\_SNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FType\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
elseif type3 == FType\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
elseif type1 == FType\_QNaN then
    done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
elseif type2 == FType\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
elseif type3 == FType\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
else
    done = FALSE; result = Zeros(32); // 'Don't care' result
return (done, result);
```



## Library pseudocode for shared/functions/float/fpmulx/FPMulX

```
// FPMulX()
// =====

bits(N) FPMulX(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    bits(N) result;
    boolean done;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPType\_Infinity);
        inf2 = (type2 == FPType\_Infinity);
        zero1 = (type1 == FPType\_Zero);
        zero2 = (type2 == FPType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2, N);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2, N);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2, N);
        else
            result = FPRound(value1*value2, fpcr, N);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

bits(N) FPNeg(bits(N) op, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    if !UsingAArch32() && IsFeatureImplemented(FEAT_AFP) then
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPType\_SNaN, FPType\_QNaN} then
                return op;
            // When fpcr.AH=1, sign of NaN has no consequence
    return NOT(op<N-1>) : op<N-2:0>;
```

## Library pseudocode for shared/functions/float/fponepointfive/FPOnePointFive

```
// FPOnePointFive()
// =====

bits(N) FPOnePointFive(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '0':Ones(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorm

```
// FPProcessDenorm()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorm(FPType fptype, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && fptype == FPType_Denormal then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms

```
// FPProcessDenorms()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms(FPType type1, FPType type2, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms3

```
// FPProcessDenorms3()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms3(FPType type1, FPType type2, FPType type3, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

## Library pseudocode for shared/functions/float/fpprocessdenorms/FPProcessDenorms4

```
// FPProcessDenorms4()
// =====
// Handles denormal input in case of single-precision or double-precision
// when using alternative floating-point mode.

FPProcessDenorms4(FPType type1, FPType type2, FPType type3, FPType type4, integer N, FPCR_Type fpcr)
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    if altfp && N != 16 && (type1 == FPType_Denormal || type2 == FPType_Denormal ||
        type3 == FPType_Denormal || type4 == FPType_Denormal) then
        FPProcessException(FPExc_InputDenorm, fpcr);
```

## 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.

FPPProcessException(FPExc except, FPCR\_Type fpcr)
    integer cumul;
    // 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;
    enable = 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().
        bits(8) accumulated_exceptions = GetAccumulatedFPEExceptions();
        accumulated_exceptions<cumul> = '1';
        if boolean IMPLEMENTATION_DEFINED "Support trapping of floating-point exceptions" then
            if UsingAArch32() then
                AArch32.FPTrappedException(accumulated_exceptions);
            else
                is_ase = IsASEInstruction();
                AArch64.FPTrappedException(is_ase, accumulated_exceptions);
        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);
    elseif UsingAArch32() then
        // Set the cumulative exception bit
        FPCSR<cumul> = '1';
    else
        // Set the cumulative exception bit
        FPSR<cumul> = '1';
    return;
```

## Library pseudocode for shared/functions/float/fpprocessnan/FPProcessNaN

```
// FPProcessNaN()
// =====

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPProcessNaN(fptype, op, fpcr, fpexc);

// 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.

bits(N) FPProcessNaN(FPType fptype, bits(N) op, FPCR_Type fpcr, boolean fpexc)
    assert N IN {16,32,64};
    assert fptype IN {FPType_QNaN, FPType_SNaN};
    integer topfrac;

    case N of
        when 16 topfrac = 9;
        when 32 topfrac = 22;
        when 64 topfrac = 51;

    result = op;
    if fptype == FPType_SNaN then
        result<topfrac> = '1';
        if fpexc then FPProcessException(FPExc_InvalidOp, fpcr);
    if fpcr.DN == '1' then // DefaultNaN requested
        result = FPDefaultNaN(fpcr, N);
    return result;
```

## Library pseudocode for shared/functions/float/fpprocessnans/FPProcessNaNs

```
// FPProcessNaNs()
// =====

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1,
                                bits(N) op2, FPCR_Type fpcr)
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);

// FPProcessNaNs()
// =====
//
// 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.

(boolean, bits(N)) FPProcessNaNs(FPType type1, FPType type2, bits(N) op1, bits(N) op2,
                                FPCR_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    boolean done;
    bits(N) result;
    constant boolean altfp      = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean op1_nan    = type1 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op2_nan    = type2 IN {FPType_SNaN, FPType_QNaN};
    constant boolean any_snan   = type1 == FPType_SNaN || type2 == FPType_SNaN;
    constant FPType   type_nan  = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result
    return (done, result);
```

## Library pseudocode for shared/functions/float/fpprocessnans3/FPProcessNaNs3

```
// FPProcessNaNs3()
// =====

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCR_Type fpcr)
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPProcessNaNs3()
// =====
// 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.

(boolean, bits(N)) FPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCR_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    constant boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    constant boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    FPType type_nan;
    if altfp then
        if type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then
        // <n> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(N); // 'Don't care' result
    return (done, result);
```

## Library pseudocode for shared/functions/float/fpprocessnans4/FPProcessNaNs4

```
// FPProcessNaNs4()
// =====
// 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.

(boolean, bits(32)) FPProcessNaNs4(FPTYPE type1, FPTYPE type2, FPTYPE type3, FPTYPE type4,
                                   bits(16) op1, bits(16) op2, bits(16) op3,
                                   bits(16) op4, FPCR\_Type fpcr, boolean fpexc)

bits(32) result;
boolean done;
// The FPCR.AH control does not affect these checks
if type1 == FPTYPE\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), 32);
elseif type2 == FPTYPE\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
elseif type3 == FPTYPE\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
elseif type4 == FPTYPE\_SNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), 32);
elseif type1 == FPTYPE\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type1, op1, fpcr, fpexc), 32);
elseif type2 == FPTYPE\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type2, op2, fpcr, fpexc), 32);
elseif type3 == FPTYPE\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type3, op3, fpcr, fpexc), 32);
elseif type4 == FPTYPE\_QNaN then
    done = TRUE; result = FPConvertNaN(FPProcessNaN(type4, op4, fpcr, fpexc), 32);
else
    done = FALSE; result = Zeros(32); // 'Don't care' result

return (done, result);
```





```

// FPrecipEstimate()
// =====

bits(N) FPrecipEstimate(bits(N) operand, FPCR\_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR\_Type fpcr = fpcr_in;
    bits(N) result;
    boolean overflow_to_inf;
    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions, flush denormal input and output to zero,
    // and use RNE rounding mode.
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';
    if altfp then fpcr.RMode = '00';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

    constant FPRounding rounding = FPRoundingMode(fpcr);
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpexc);
    elseif fptype == FPTType\_Infinity then
        result = FPZero(sign, N);
    elseif fptype == FPTType\_Zero then
        result = FPInfinity(sign, N);
        if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
    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;
        result = if overflow_to_inf then FPInfinity(sign, N) else FPMaxNormal(sign, N);
        if fpexc then
            FPProcessException(FPExc\_Overflow, fpcr);
            FPProcessException(FPExc\_Inexact, fpcr);
    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(sign, N);

        // Flush-to-zero never generates a trapped exception.
        if UsingAArch32() then
            FPSCR.UFC = '1';
        else
            if fpexc then FPSCR.UFC = '1';
    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
        bits(52) fraction;
        integer exp;
        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>);

    if exp == 0 then
        if fraction<51> == '0' then
            exp = -1;
            fraction = fraction<49:0>:'00';
        else
            fraction = fraction<50:0>:'0';

    integer scaled;
    constant boolean increasedprecision = N==32 && IsFeatureImplemented(FEAT_RPRES) && altfp;

    if !increasedprecision then
        scaled = UInt('1':fraction<51:44>);
    else
        scaled = UInt('1':fraction<51:41>);

    integer result_exp;
    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

    // Scaled is in range 256 .. 511 or 2048 .. 4095 range representing a
    // fixed-point number in range [0.5 .. 1.0].
    estimate = 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);

    if result_exp == 0 then
        fraction = '1' : fraction<51:1>;
    elsif result_exp == -1 then
        fraction = '01' : fraction<51:2>;
        result_exp = 0;

    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>;

    return result;

```

## 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.

integer RecipEstimate(integer a_in, boolean increasedprecision)
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 256 <= a && a < 512;
        a = a*2+1; // Round to nearest
        constant integer b = (2 ^ 19) DIV a;
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 2048 <= a && a < 4096;
        a = a*2+1; // Round to nearest
        constant real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        constant real error = real_val - Real(r);
        constant boolean round_up = error > 0.5; // Error cannot be exactly 0.5 so do not
                                                // need tie case
        if round_up then r = r+1;
        assert 4096 <= r && r < 8192;
    return r;
```

## Library pseudocode for shared/functions/float/fprecpX/FPRecpX

```
// FPRecpX()
// =====

bits(N) FPRecpX(bits(N) op, FPCR\_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR\_Type fpcr = fpcr_in;
    constant boolean isbfloat16 = FALSE;
    constant (F, -) = FPBits(N, isbfloat16);
    constant E = (N - F) - 1;
    bits(N) result;
    bits(E) exp;
    bits(E) max_exp;
    constant bits(F) frac = Zeros(F);

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && fpcr.AH == '1';
    constant boolean fpexc = !altfp; // Generate no floating-point exceptions
    if altfp then fpcr.<FIZ,FZ> = '11'; // Flush denormal input and output to zero
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);
    exp = op<F+:E>;
    max_exp = Ones(E) - 1;

    if fptype == FType\_SNaN || fptype == FType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then // Zero and denormals
            result = ZeroExtend(sign:max_exp:frac, N);
        else // Infinities and normals
            result = ZeroExtend(sign:NOT(exp):frac, N);

    return result;
```

## Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =====
// Generic conversion from precise, unbounded real data type to IEEE format.

bits(N) FPRound(real op, FPCR\_Type fpcr, integer N)
    return FPRound(op, fpcr, FPRoundingMode(fpcr), N);

// FPRound()
// =====
// For directed FP conversion, includes an explicit 'rounding' argument.

bits(N) FPRound(real op, FPCR\_Type fpcr_in, FPRounding rounding, integer N)
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    return FPRound(op, fpcr_in, rounding, fpexc, N);

// FPRound()
// =====
// For AltFP, includes an explicit FPEXC argument to disable exception
// generation and switches off Arm alternate half-precision mode.

bits(N) FPRound(real op, FPCR\_Type fpcr_in, FPRounding rounding, boolean fpexc, integer N)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);
```



```

// FPRoundBase()
// =====
// For BFloat16, includes an explicit 'isbfloat16' argument.

bits(N) FPRoundBase(real op, FPCR\_Type fpcr, FPRounding rounding, boolean isbfloat16, integer N)
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

// FPRoundBase()
// =====
// For FP8 multiply-accumulate, dot product, and outer product instructions, includes
// an explicit saturation overflow argument.

bits(N) FPRoundBase(real op, FPCR\_Type fpcr, FPRounding rounding, boolean isbfloat16,
    boolean fpexc, integer N)
    constant boolean satoflo = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, satoflo, N);

// 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.

bits(N) FPRoundBase(real op, FPCR\_Type fpcr, FPRounding rounding, boolean isbfloat16,
    boolean fpexc, boolean satoflo, integer N)

    assert N IN {16,32,64};
    assert op != 0.0;
    assert rounding != FPRounding\_TIEAWAY;
    bits(N) result;

    // Obtain format parameters - minimum exponent, numbers of exponent and fraction bits.
    constant (F, minimum_exp) = FPBits(N, isbfloat16);
    constant zeros = if N == 32 && isbfloat16 then 16 else 0;
    constant E = N - (F + 1 + zeros);
    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    integer exponent;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    (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.
    altfp = 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
            FPSR.UFC = '1';
        else
            if fpexc then FPSR.UFC = '1';
        return FPZero(sign, N);

    biased_exp_unconstrained = (exponent - minimum_exp) + 1;
    int_mant_unconstrained = RoundDown(mantissa * 2.0^F);
    error_unconstrained = 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).
biased_exp = Max((exponent - minimum_exp) + 1, 0);
if biased_exp == 0 then mantissa = mantissa / 2.0^(minimum_exp - exponent);

// Get the unrounded mantissa as an integer, and the "units in last place" rounding error.
int_mant = RoundDown(mantissa * 2.0^F); // < 2.0^F if biased_exp == 0, >= 2.0^F if not
error = 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'.
constant boolean trapped_UF = fpcr.UFE == '1' && (!InStreamingMode() || IsFullA64Enabled());
if !altfp && biased_exp == 0 && (error != 0.0 || trapped_UF) then
    if fpexc then FPProcessException(FPExc Underflow, fpcr);

// Round result according to rounding mode.
boolean round_up_unconstrained;
boolean round_up;
boolean overflow_to_inf;
if altfp then

    case rounding of
        when FPRounding TIEVEN
            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;

    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;

// 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';
            FPProcessException(FPExc Inexact, fpcr);
            return FPZero(sign, N);
        elsif error != 0.0 || trapped_UF then
            if fpexc then FPProcessException(FPExc Underflow, fpcr);
else // altfp == FALSE
    case rounding of
        when FPRounding TIEVEN
            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;

```

```

if round_up then
    int_mant = int_mant + 1;
    if int_mant == 2^F then          // Rounded up from denormalized to normalized
        biased_exp = 1;
    if int_mant == 2^(F+1) then      // Rounded up to next exponent
        biased_exp = biased_exp + 1;
        int_mant = int_mant DIV 2;

// Handle rounding to odd
if error != 0.0 && rounding == FPRounding\_ODD && int_mant<0> == '0' then
    int_mant = int_mant + 1;

// 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(sign, N) else FPMMaxNormal(sign, N);
        if fpexc then FPProcessException(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> : Zeros(N-(E+F+1));
else
    // Alternative half precision
    if biased_exp >= 2^E then
        result = sign : Ones(N-1);
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        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));

// Deal with Inexact exception.
if error != 0.0 then
    if fpexc then FPProcessException(FPExc\_Inexact, fpcr);

return result;

```

### 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.

bits(N) FPRoundCV(real op, FPCR\_Type fpcr_in, FPRounding rounding, integer N)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, N);

```

### 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.

bits(N) FPRound_FP8(real op, FPCR\_Type fpcr_in, FPRounding rounding,
    boolean satoflo, integer N)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean fpexc = FALSE;
    constant boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc, satoflo, N);

```



## 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.

enumeration FPRounding {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.

FPRounding FPRoundingMode(FPCR\_Type fpcr)
    return FPDecodeRounding(fpcr.RMode);
```

## 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.

bits(N) FPRoundInt(bits(N) op, FPCR\_Type fpcr, FPRounding rounding, boolean exact)
    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.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    bits(N) result;
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTType\_Infinity then
        result = FPInfinity(sign, N);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        error = value - Real(int_result);

        // Determine whether supplied rounding mode requires an increment.
        boolean round_up;
        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));

        if round_up then int_result = int_result + 1;

        // Convert integer value into an equivalent real value.
        real_result = Real(int_result);

        // Re-encode as a floating-point value, result is always exact.
        if real_result == 0.0 then
            result = FPZero(sign, N);
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO, N);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```



```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCR\_Type fpcr, FPRounding rounding, integer intsize)
    assert rounding != FPRounding\_ODD;
    assert N IN {32,64};
    assert intsize IN {32, 64};
    integer exp;
    bits(N) result;
    boolean round_up;
    constant integer E = (if N == 32 then 8 else 11);
    constant integer F = N - (E + 1);

    // When alternative floating-point support is TRUE, do not generate
    // Input Denormal floating-point exceptions.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using FPCR to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    if fptype IN {FPTType\_SNaN, FPType\_QNaN, FPTType\_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);
            FPProcessException(FPExc\_InvalidOp, fpcr);
    elseif fptype == FPTType\_Zero then
        result = FPZero(sign, N);
    else
        // Extract integer component.
        int_result = RoundDown(value);
        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);

        if round_up then int_result = int_result + 1;
        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);
                FPProcessException(FPExc\_InvalidOp, fpcr);
                // This case shouldn't set Inexact.
                error = 0.0;

        else
            // Convert integer value into an equivalent real value.
            real_result = Real(int_result);

            // Re-encode as a floating-point value, result is always exact.
            if real_result == 0.0 then
                result = FPZero(sign, N);

```

```
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO, N);

// Generate inexact exceptions.
if error != 0.0 then
    FPProcessException(FPExc\_Inexact, fpcr);

return result;
```



```

// FPRSqrtEstimate()
// =====

bits(N) FPRSqrtEstimate(bits(N) operand, FPCR\_Type fpcr_in)
    assert N IN {16,32,64};
    FPCR\_Type fpcr = fpcr_in;

    // When using alternative floating-point behavior, do not generate
    // floating-point exceptions and flush denormal input to zero.
    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    constant boolean fpexc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpexc);

bits(N) result;
if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpexc);
elseif fptype == FPTType\_Zero then
    result = FPInfinity(sign, N);
    if fpexc then FPProcessException(FPExc\_DivideByZero, fpcr);
elseif sign == '1' then
    result = FPDefaultNaN(fpcr, N);
    if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
elseif fptype == FPTType\_Infinity then
    result = FPZero('0', N);
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.

bits(52) fraction;
integer exp;
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>);

if exp == 0 then
    while fraction<51> == '0' do
        fraction = fraction<50:0> : '0';
        exp = exp - 1;
    fraction = fraction<50:0> : '0';

integer scaled;
constant boolean increasedprecision = 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>);
else
    if exp<0> == '0' then
        scaled = UInt('1':fraction<51:41>);
    else
        scaled = UInt('01':fraction<51:42>);

integer result_exp;
case N of
    when 16 result_exp = ( 44 - exp) DIV 2;
    when 32 result_exp = ( 380 - exp) DIV 2;
    when 64 result_exp = (3068 - exp) DIV 2;

```

```

estimate = 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);
  when 64 result = '0' : result_exp<N-54:0> : estimate<7:0>:Zeros(44);

return result;

```



## Library pseudocode for shared/functions/float/fpsqrtestimate/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.

integer RecipSqrtEstimate(integer a_in, boolean increasedprecision)
    integer a = a_in;
    integer r;
    if !increasedprecision then
        assert 128 <= a && a < 512;
        if a < 256 then // 0.25 .. 0.5
            a = a*2+1; // a in units of 1/512 rounded to nearest
        else // 0.5 .. 1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = (a+1)*2; // a in units of 1/256 rounded to nearest
        integer b = 512;
        while a*(b+1)*(b+1) < 2^28 do
            b = b+1;
        // b = largest b such that b < 2^14 / sqrt(a)
        r = (b+1) DIV 2; // Round to nearest
        assert 256 <= r && r < 512;
    else
        assert 1024 <= a && a < 4096;
        real real_val;
        real error;
        integer int_val;

        if a < 2048 then // 0.25... 0.5
            a = a*2 + 1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a)/2.0;
        else // 0.5..1.0
            a = (a >> 1) << 1; // Discard bottom bit
            a = a+1; // Take 10 bits of fraction and force a 1 at the bottom
            real_val = Real(a);
        // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        (real_val, -) = SqrtRoundDown(real_val, 54);
        real_val = real_val * Real(2^47); // The integer is the size of the whole DP mantissa
        int_val = RoundDown(real_val); // Calculate rounding value
        error = real_val - Real(int_val);
        round_up = error > 0.5; // Error cannot be exactly 0.5 so do not need tie case
        if round_up then int_val = int_val+1;

        real_val = Real(2^65)/Real(int_val); // Lies in the range 4096 <= real_val < 8192
        int_val = RoundDown(real_val); // Round that (to nearest even) to give integer
        error = real_val - Real(int_val);
        round_up = (error > 0.5 || (error == 0.5 && int_val<0> == '1'));
        if round_up then int_val = int_val+1;

        r = int_val;
        assert 4096 <= r && r < 8192;
    return r;
```

## Library pseudocode for shared/functions/float/fpsqrt/FPSqrt

```
// FPSqrt()
// =====

bits(N) FPSqrt(bits(N) op, FPCR\_Type fpcr)
    assert N IN {16,32,64};
    (fptype,sign,value) = FPUnpack(op, fpcr);

    bits(N) result;
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero(sign, N);
    elsif fptype == FPTYPE\_Infinity && sign == '0' then
        result = FPInfinity(sign, N);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr, N);
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        integer prec;
        boolean inexact;
        if N == 16 then
            prec = 12; // 10 fraction bit + 2
        elsif N == 32 then
            prec = 25; // 23 fraction bits + 2
        else // N == 64
            prec = 54; // 52 fraction bits + 2
        (value, inexact) = SqrtRoundDown(value, prec);
        result = FPRound(value, fpcr, N);
        if inexact then
            FPProcessException(FPExc\_Inexact, fpcr);
            FPProcessDenorm(fptype, N, fpcr);

    return result;
```

## Library pseudocode for shared/functions/float/fpsub/FPSub

```
// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCR\_Type fpcr)
    constant boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPSub(op1, op2, fpcr, fpexc);

// FPSub()
// =====

bits(N) FPSub(bits(N) op1, bits(N) op2, FPCR\_Type fpcr, boolean fpexc)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUnpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUnpack(op2, fpcr, fpexc);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, fpexc);
    if !done then
        inf1 = (type1 == FPTYPE\_Infinity);
        inf2 = (type2 == FPTYPE\_Infinity);
        zero1 = (type1 == FPTYPE\_Zero);
        zero2 = (type2 == FPTYPE\_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr, N);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0', N);
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1', N);
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1, N);
        else
            result_value = value1 - value2;
            if result_value == 0.0 then // Sign of exact zero result depends on rounding mode
                result_sign = if rounding == FPRounding\_NEGINF then '1' else '0';
                result = FPZero(result_sign, N);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc, N);

            if fpexc then FPProcessDenorms(type1, type2, N, fpcr);
    return result;
```

## Library pseudocode for shared/functions/float/fpsub/FPSub\_ZA

```
// FPSub_ZA()
// =====
// Calculates op1-op2 for SME2 ZA-targeting instructions.

bits(N) FPSub_ZA(bits(N) op1, bits(N) op2, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    constant boolean fpexc = FALSE; // Do not generate floating-point exceptions
    fpcr.DN = '1'; // Generate default NaN values
    return FPSub(op1, op2, fpcr, fpexc);
```

## Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

bits(N) FPThree(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = '1':Zeros(E-1);
    frac = '1':Zeros(F-1);
    result = sign : exp : frac;

    return result;
```

## 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.

bits(M) FPToFixed(bits(N) op, integer fbits, boolean unsigned, FPCR\_Type fpcr,
FPRounding rounding, integer M)
    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.
    altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1';
    fpexc = !altfp;

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUnpack(op, fpcr, fpexc);

    // If NaN, set cumulative flag or take exception.
    if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);

    // Scale by fractional bits and produce integer rounded towards minus-infinity.
    value = value * 2.0^fbits;
    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.
    boolean round_up;
    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));

    if round_up then int_result = int_result + 1;

    // Generate saturated result and exceptions.
    (result, overflow) = SatQ(int_result, M, unsigned);
    if overflow then
        FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif error != 0.0 then
        FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```

## 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.

(bits(N), bit) FPToFixedJS(bits(M) op, FPCR\_Type fpcr, integer N)
    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpexc_idenorm = !(IsFeatureImplemented(FEAT_AFP) && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype, sign, value) = FPUnpack(op, fpcr, fpexc_idenorm);

    z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPType\_SNaN || fptype == FPType\_QNaN then
        FPProcessException(FPExc\_InvalidOp, fpcr);
        z = '0';

    int_result = RoundDown(value);
    error = value - Real(int_result);

    // Determine whether supplied rounding mode requires an increment.

    round_it_up = (error != 0.0 && int_result < 0);
    if round_it_up then int_result = int_result + 1;

    integer result;
    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));

    // 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';

    if fptype == FPType\_Infinity then result = 0;

    return (result<N-1:0>, z);
```

## Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign, integer N)
    assert N IN {16, 32, 64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp = '1':Zeros(E-1);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

## Library pseudocode for shared/functions/float/fptype/FPType

```
// FPType
// =====

enumeration FPType {FPType_Zero,
                    FPType_Denormal,
                    FPType_Nonzero,
                    FPType_Infinity,
                    FPType_QNaN,
                    FPType_SNaN};
```

## Library pseudocode for shared/functions/float/fpunpack/FPUnpack

```
// FPUnpack()
// =====

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// 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.

(FPType, bit, real) FPUnpack(bits(N) fpval, FPCR\_Type fpcr_in, boolean fpexc)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.AHP = '0';
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);
```





```

// FPUnpackBase()
// =====

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCR\_Type fpcr, boolean fpexc)
    constant boolean isbfloat16 = FALSE;
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc, isbfloat16);
    return (fp_type, sign, value);

// 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.

(FPType, bit, real) FPUnpackBase(bits(N) fpval, FPCR\_Type fpcr_in, boolean fpexc,
                                boolean isbfloat16)
    assert N IN {16,32,64};

    constant FPCR\_Type fpcr = fpcr_in;

    constant boolean altfp = IsFeatureImplemented(FEAT_AFP) && !UsingAArch32();
    constant boolean fiz   = altfp && fpcr.FIZ == '1';
    constant boolean fz    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 && !isbfloat16 then
        sign    = fpval<15>;
        exp16   = fpval<14:10>;
        frac16  = fpval<9:0>;
        if IsZero(exp16) then
            if IsZero(frac16) || fpcr.FZ16 == '1' then
                fptype = FPType\_Zero; value = 0.0;
            else
                fptype = FPType\_Denormal; value = 2.0^-14 * (Real(UInt(frac16)) * 2.0^-10);
        elsif IsOnes(exp16) && fpcr.AHP == '0' then // Infinity or NaN in IEEE format
            if IsZero(frac16) then
                fptype = FPType\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac16<9> == '1' then FPType\_QNaN else FPType\_SNaN;
                value = 0.0;
        else
            fptype = FPType\_Nonzero;
            value = 2.0^(UInt(exp16)-15) * (1.0 + Real(UInt(frac16)) * 2.0^-10);

    elsif N == 32 || isbfloat16 then
        bits(8) exp32;
        bits(23) frac32;
        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>;

        if IsZero(exp32) then
            if IsZero(frac32) 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);
        else
            fptype = FPTYPE\_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
    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;
    else
        fptype = FPTYPE\_Nonzero;
        value = 2.0^(UInt(exp32)-127) * (1.0 + Real(UInt(frac32)) * 2.0^-23);

else // N == 64
    sign    = fpval<63>;
    exp64   = fpval<62:52>;
    frac64  = 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);
            else
                fptype = FPTYPE\_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        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;
        else
            fptype = FPTYPE\_Nonzero;
            value = 2.0^(UInt(exp64)-1023) * (1.0 + Real(UInt(frac64)) * 2.0^-52);

    if sign == '1' then value = -value;

    return (fptype, sign, value);

```

## 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.

(FPType, bit, real) FPUnpackCV(bits(N) fpval, FPCR\_Type fpcr_in)
    FPCR\_Type fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    constant boolean fpexc = TRUE;    // Generate floating-point exceptions
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

```

## Library pseudocode for shared/functions/float/fpzero/FPZero

```
// FPZero()
// =====

bits(N) FPZero(bit sign, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = N - (E + 1);
    exp  = Zeros(E);
    frac = Zeros(F);
    result = sign : exp : frac;
    return result;
```

## Library pseudocode for shared/functions/float/vfpexpandimm/VFPEExpandImm

```
// VFPEExpandImm()
// =====

bits(N) VFPEExpandImm(bits(8) imm8, integer N)
    assert N IN {16,32,64};
    constant integer E = (if N == 16 then 5 elsif N == 32 then 8 else 11);
    constant integer F = (N - E) - 1;
    sign = imm8<7>;
    exp  = NOT(imm8<6>):Replicate(imm8<6>,E-3):imm8<5:4>;
    frac = imm8<3:0>:Zeros(F-4);
    result = sign : exp : frac;

    return result;
```

## Library pseudocode for shared/functions/integer/AddWithCarry

```
// AddWithCarry()
// =====
// Integer addition with carry input, returning result and NZCV flags

(bits(N), bits(4)) AddWithCarry(bits(N) x, bits(N) y, bit carry_in)
    constant integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    constant integer signed_sum  = SInt(x) + SInt(y) + UInt(carry_in);
    constant bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    constant bit n = result<N-1>;
    constant bit z = if IsZero(result) then '1' else '0';
    constant bit c = if UInt(result) == unsigned_sum then '0' else '1';
    constant bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

## Library pseudocode for shared/functions/interrupts/InterruptID

```
// InterruptID
// =====

enumeration InterruptID {
    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.

SetInterruptRequestLevel(InterruptID id, Signal level);
```

## 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.

bits(64) AArch64.BranchAddr(bits(64) vaddress, bits(2) el)
    assert !UsingAArch32();
    constant integer msbit = AddrTop(vaddress, TRUE, el);
    if msbit == 63 then
        return vaddress;
    elsif (el IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then
        return SignExtend(vaddress<msbit:0>, 64);
    else
        return ZeroExtend(vaddress<msbit:0>, 64);
```

## Library pseudocode for shared/functions/memory/AccessDescriptor

```
// AccessDescriptor
// =====
// Memory access or translation invocation details that steer architectural behavior

type AccessDescriptor is (
    AccessType acctype,
    bits(2) el, // Acting EL for the access
    SecurityState ss, // Acting Security State for the access
    boolean acqsc, // Acquire with Sequential Consistency
    boolean acqpc, // FEAT_LRCPC: Acquire with Processor Consistency
    boolean relsc, // Release with Sequential Consistency
    boolean limitedordered, // FEAT_LOR: Acquire/Release with limited ordering
    boolean exclusive, // Access has Exclusive semantics
    boolean atomicop, // FEAT_LSE: Atomic read-modify-write access
    MemAtomicOp modop, // FEAT_LSE: The modification operation in the 'atomicop' access
    boolean nontemporal, // Hints the access is non-temporal
    boolean read, // Read from memory or only require read permissions
    boolean write, // Write to memory or only require write permissions
    CacheOp cacheop, // DC/IC: Cache operation
    CacheOpScope opscope, // DC/IC: Scope of cache operation
    CacheType cachetype, // DC/IC: Type of target cache
    boolean pan, // FEAT_PAN: The access is subject to PSTATE.PAN
    boolean transactional, // FEAT_TME: Access is part of a transaction
    boolean nonfault, // SVE: Non-faulting load
    boolean firstfault, // SVE: First-fault load
    boolean first, // SVE: First-fault load for the first active element
    boolean contiguous, // SVE: Contiguous load/store not gather load/scatter store
    boolean streamingsve, // SME: Access made by PE while in streaming SVE mode
    boolean ls64, // FEAT_LS64: Accesses by accelerator support loads/stores
    boolean withstatus, // FEAT_LS64: Store with status result
    boolean mops, // FEAT_MOPS: Memory operation (CPY/SET) accesses
    boolean rcw, // FEAT_THE: Read-Check-Write access
    boolean rcws, // FEAT_THE: Read-Check-Write Software access
    boolean toplevel, // FEAT_THE: Translation table walk access for TTB address
    VARange varange, // FEAT_THE: The corresponding TTBR supplying the TTB
    boolean a32lsmd, // A32 Load/Store Multiple Data access
    boolean tagchecked, // FEAT_MTE2: Access is tag checked
    boolean tagaccess, // FEAT_MTE: Access targets the tag bits
    boolean stzgm, // FEAT_MTE: Accesses that store Allocation tags to Device
    // memory are CONSTRAINED UNPREDICTABLE
    boolean ispair, // Access represents a Load/Store pair access
    boolean highestaddressfirst, // FEAT_LRCPC3: Highest address is accessed first
    MPAMInfo mpam // FEAT_MPAM: MPAM information
)
```

## Library pseudocode for shared/functions/memory/AccessType

```
// AccessType
// =====

enumeration AccessType {
    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.

AddressSize AddrTop(bits(64) address, boolean IsInstr, bits(2) el)
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    if ELUsingAArch32(regime) then
        // AArch32 translation regime.
        return 31;
    else
        if EffectiveTBI(address, IsInstr, el) == '1' then
            return 55;
        else
            return 63;
```

## Library pseudocode for shared/functions/memory/AddressSize

```
// AddressSize
// =====

type AddressSize = integer;
```

## Library pseudocode for shared/functions/memory/AlignmentEnforced

```
// AlignmentEnforced()
// =====
// For the active translation regime, determine if alignment is required by all accesses

boolean AlignmentEnforced()
    bit A;
    constant Regime regime = TranslationRegime(PSTATE.EL);
    case regime of
        when Regime\_EL3 A = SCTLR_EL3.A;
        when Regime\_EL30 A = SCTLR.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();
    return A == '1';
```

## Library pseudocode for shared/functions/memory/Allocation

```
// Allocation hints
// =====

constant bits(2) MemHint_No = '00'; // No Read-Allocate, No Write-Allocate
constant bits(2) MemHint_WA = '01'; // No Read-Allocate, Write-Allocate
constant bits(2) MemHint_RA = '10'; // Read-Allocate, No Write-Allocate
constant bits(2) MemHint_RWA = '11'; // Read-Allocate, Write-Allocate
```

## Library pseudocode for shared/functions/memory/BigEndian

```
// BigEndian()
// =====

boolean BigEndian(AccessType acctype)
    boolean bigend;
    if IsFeatureImplemented(FEAT_NV2) && acctype == AccessType\_NV2 then
        return SCTLR_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR_ELx[].EOE != '0');
    else
        bigend = (SCTLR_ELx[].EE != '0');
    return bigend;
```

## Library pseudocode for shared/functions/memory/BigEndianReverse

```
// BigEndianReverse()
// =====

bits(width) BigEndianReverse(bits(width) value)
    assert width IN {8, 16, 32, 64, 128, 256};

    if width == 8 then return value;
    return Reverse(value, 8);
```

## Library pseudocode for shared/functions/memory/CacheOp

```
// CacheOp
// =====

enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

## Library pseudocode for shared/functions/memory/CacheOpScope

```
// CacheOpScope
// =====

enumeration CacheOpScope {
    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
// =====

enumeration CachePASpace {
    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
// =====

enumeration CacheType {
    CacheType_Data,
    CacheType_Tag,
    CacheType_Data_Tag,
    CacheType_Instruction
};
```

## Library pseudocode for shared/functions/memory/Cacheability

```
// Cacheability attributes
// =====

constant bits(2) MemAttr_NC = '00';    // Non-cacheable
constant bits(2) MemAttr_WT = '10';    // Write-through
constant bits(2) MemAttr_WB = '11';    // Write-back
```



## Library pseudocode for shared/functions/memory/CreateAccDescA32LSMD

```
// CreateAccDescA32LSMD()
// =====
// Access descriptor for A32 loads/store multiple general purpose registers

AccessDescriptor CreateAccDescA32LSMD(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.read          = memop == MemOp_LOAD;
    accdesc.write         = memop == MemOp_STORE;
    accdesc.pan           = TRUE;
    accdesc.a32lsmd       = TRUE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescASIMD

```
// CreateAccDescASIMD()
// =====
// Access descriptor for ASIMD&FP loads/stores

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked,
                                     boolean privileged)
    constant boolean ispair = FALSE;
    return CreateAccDescASIMD(memop, nontemporal, tagchecked, privileged, ispair);

// CreateAccDescASIMD()
// =====

AccessDescriptor CreateAccDescASIMD(MemOp memop, boolean nontemporal, boolean tagchecked,
                                     boolean privileged, boolean ispair)
    AccessDescriptor accdesc = NewAccDesc(AccessType_ASIMD);

    accdesc.nontemporal    = nontemporal;
    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.streamingsve   = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;
    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescASIMDAcqRel

```
// CreateAccDescASIMDAcqRel()
// =====
// Access descriptor for ASIMD&FP loads/stores with ordering semantics

AccessDescriptor CreateAccDescASIMDAcqRel(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = 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.streaming sve  = InStreamingMode();
    if (accdesc.streaming sve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAT

```
// CreateAccDescAT()
// =====
// Access descriptor for address translation operations

AccessDescriptor CreateAccDescAT(SecurityState ss, bits(2) el, ATAccess ataccess)
    AccessDescriptor accdesc = NewAccDesc(AccessType_AT);

    accdesc.el          = el;
    accdesc.ss          = ss;
    if boolean IMPLEMENTATION_DEFINED "MPAM uses the EL targeted by the AT instruction" then
        accdesc.mpam = GenMPAMAtEL(AccessType_AT, el);
    case ataccess of
        when ATAccess_Read
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, FALSE);
        when ATAccess_ReadPAN
            (accdesc.read, accdesc.write, accdesc.pan) = (TRUE, FALSE, TRUE);
        when ATAccess_Write
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, FALSE);
        when ATAccess_WritePAN
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, TRUE, TRUE);
        when ATAccess_Any
            (accdesc.read, accdesc.write, accdesc.pan) = (FALSE, FALSE, FALSE);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAcqRel

```
// CreateAccDescAcqRel()
// =====
// Access descriptor for general purpose register loads/stores with ordering semantics

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked)
    constant boolean ispair = FALSE;
    return CreateAccDescAcqRel(memop, tagchecked, ispair);

AccessDescriptor CreateAccDescAcqRel(MemOp memop, boolean tagchecked, boolean ispair)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc          = memop == MemOp\_LOAD;
    accdesc.relsc          = memop == MemOp\_STORE;
    accdesc.read           = memop == MemOp\_LOAD;
    accdesc.write          = memop == MemOp\_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;
    accdesc.ispair         = ispair;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescAtomicOp

```
// CreateAccDescAtomicOp()
// =====
// Access descriptor for atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                       boolean tagchecked, boolean privileged)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqsc          = acquire;
    accdesc.el             = if !privileged then EL0 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.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescDC

```
// CreateAccDescDC()
// =====
// Access descriptor for data cache operations

AccessDescriptor CreateAccDescDC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_DC);

    accdesc.cacheop        = cache.cacheop;
    accdesc.cachetype       = cache.cachetype;
    accdesc.opscope        = cache.opscope;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescDCZero

```
// CreateAccDescDCZero()
// =====
// Access descriptor for data cache zero operations

AccessDescriptor CreateAccDescDCZero(CacheType cachetype)
    AccessDescriptor accdesc = 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;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescExLDST

```
// CreateAccDescExLDST()
// =====
// Access descriptor for general purpose register loads/stores with exclusive semantics

AccessDescriptor CreateAccDescExLDST(MemOp memop, boolean acqrel, boolean tagchecked,
                                     boolean privileged)
    AccessDescriptor accdesc = 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.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescFPAtomicOp

```
// CreateAccDescFPAtomicOp()
// =====
// Access descriptor for FP atomic read-modify-write memory accesses

AccessDescriptor CreateAccDescFPAtomicOp(MemAtomicOp modop, boolean acquire, boolean release,
                                         boolean tagchecked)
    AccessDescriptor accdesc = 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 && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGCS

```
// CreateAccDescGCS()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack

AccessDescriptor CreateAccDescGCS(MemOp memop, boolean privileged)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GCS);

    accdesc.el          = if !privileged then EL0 else PSTATE.EL;
    accdesc.read        = memop == MemOp\_LOAD;
    accdesc.write       = memop == MemOp\_STORE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGCSSS1

```
// CreateAccDescGCSSS1()
// =====
// Access descriptor for memory accesses to the Guarded Control Stack that switch stacks

AccessDescriptor CreateAccDescGCSSS1(boolean privileged)
    AccessDescriptor accdesc = 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;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGPR

```
// CreateAccDescGPR()
// =====
// Access descriptor for general purpose register loads/stores
// without exclusive or ordering semantics

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                boolean tagchecked)
    constant boolean ispair = FALSE;
    return CreateAccDescGPR(memop, nontemporal, privileged, tagchecked, ispair);

AccessDescriptor CreateAccDescGPR(MemOp memop, boolean nontemporal, boolean privileged,
                                boolean tagchecked, boolean ispair)
    AccessDescriptor accdesc = 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.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;
    accdesc.ispair      = ispair;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescGPTW

```
// CreateAccDescGPTW()
// =====
// Access descriptor for Granule Protection Table walks

AccessDescriptor CreateAccDescGPTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescHACDBS

```
// CreateAccDescHACDBS()
// =====
// Access descriptor for memory accesses to the HACDBS structure.

AccessDescriptor CreateAccDescHACDBS()
    AccessDescriptor accdesc = NewAccDesc(AccessType\_HACDBS);

    accdesc.read        = TRUE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescHDBSS

```
// CreateAccDescHDBSS()
// =====
// Access descriptor for appending entries to the HDBSS

AccessDescriptor CreateAccDescHDBSS(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_HDBSS);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.write        = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescIC

```
// CreateAccDescIC()
// =====
// Access descriptor for instruction cache operations

AccessDescriptor CreateAccDescIC(CacheRecord cache)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_IC);

    accdesc.cacheop      = cache.cacheop;
    accdesc.cachetype    = cache.cachetype;
    accdesc.opscope      = cache.opscope;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescIFetch

```
// CreateAccDescIFetch()
// =====
// Access descriptor for instruction fetches

AccessDescriptor CreateAccDescIFetch()
    constant AccessDescriptor accdesc = NewAccDesc(AccessType\_IFETCH);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLDAcqPC

```
// CreateAccDescLDAcqPC()
// =====
// Access descriptor for general purpose register loads with local ordering semantics

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked)
    constant boolean ispair = FALSE;
    return CreateAccDescLDAcqPC(tagchecked, ispair);

AccessDescriptor CreateAccDescLDAcqPC(boolean tagchecked, boolean ispair)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.acqpc          = TRUE;
    accdesc.read           = TRUE;
    accdesc.pan            = TRUE;
    accdesc.tagchecked     = tagchecked;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;
    accdesc.ispair         = ispair;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLDGSTG

```
// CreateAccDescLDGSTG()
// =====
// Access descriptor for tag memory loads/stores

AccessDescriptor CreateAccDescLDGSTG(MemOp memop, boolean stzgm)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_GPR);

    accdesc.read           = memop == MemOp\_LOAD;
    accdesc.write          = memop == MemOp\_STORE;
    accdesc.pan            = TRUE;
    accdesc.tagaccess      = TRUE;
    accdesc.stzgm          = stzgm;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLOR

```
// CreateAccDescLOR()
// =====
// Access descriptor for general purpose register loads/stores with limited ordering semantics

AccessDescriptor CreateAccDescLOR(MemOp memop, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.acqsc          = memop == MemOp_LOAD;
    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.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescLS64

```
// CreateAccDescLS64()
// =====
// Access descriptor for accelerator-supporting memory accesses

AccessDescriptor CreateAccDescLS64(MemOp memop, boolean withstatus, boolean tagchecked)
    AccessDescriptor accdesc = 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;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;
    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescMOPS

```
// CreateAccDescMOPS()
// =====
// Access descriptor for data memory copy and set instructions

AccessDescriptor CreateAccDescMOPS(MemOp memop, boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = 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;
    accdesc.transactional  = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```



## Library pseudocode for shared/functions/memory/CreateAccDescNV2

```
// CreateAccDescNV2()
// =====
// Access descriptor nested virtualization memory indirection loads/stores

AccessDescriptor CreateAccDescNV2(MemOp memop)
    AccessDescriptor accdesc = NewAccDesc(AccessType_NV2);

    accdesc.el          = EL2;
    accdesc.ss          = SecurityStateAtEL(EL2);
    accdesc.read        = memop == MemOp_LOAD;
    accdesc.write       = memop == MemOp_STORE;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescRCW

```
// CreateAccDescRCW()
// =====
// Access descriptor for atomic read-check-write memory accesses

AccessDescriptor CreateAccDescRCW(MemAtomicOp modop, boolean soft, boolean acquire,
                                boolean release, boolean tagchecked)
    AccessDescriptor accdesc = 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  = IsFeatureImplemented(FEAT_MTE2) && tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescS1TTW

```
// CreateAccDescS1TTW()
// =====
// Access descriptor for stage 1 translation table walks

AccessDescriptor CreateAccDescS1TTW(boolean toplevel, VRange varange, AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.toplevel    = toplevel;
    accdesc.varange     = varange;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescS2TTW

```
// CreateAccDescS2TTW()
// =====
// Access descriptor for stage 2 translation table walks

AccessDescriptor CreateAccDescS2TTW(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_TTW);

    accdesc.el          = accdesc_in.el;
    accdesc.ss          = accdesc_in.ss;
    accdesc.read        = TRUE;
    accdesc.mpam        = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSME

```
// CreateAccDescSME()
// =====
// Access descriptor for SME loads/stores

AccessDescriptor CreateAccDescSME(MemOp memop, boolean nontemporal, boolean contiguous,
                                boolean tagchecked)
    AccessDescriptor accdesc = 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 boolean IMPLEMENTATION_DEFINED "No tag checking of SME LDR & STR instructions" then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSPE

```
// CreateAccDescSPE()
// =====
// Access descriptor for memory accesses by Statistical Profiling unit

AccessDescriptor CreateAccDescSPE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_SPE);

    accdesc.el          = owning_el;
    accdesc.ss          = owning_ss;
    accdesc.write       = TRUE;
    accdesc.mpam        = GenMPAMAtEL(AccessType\_SPE, owning_el);

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSTGMOPS

```
// CreateAccDescSTGMOPS()
// =====
// Access descriptor for tag memory set instructions

AccessDescriptor CreateAccDescSTGMOPS(boolean privileged, boolean nontemporal)
    AccessDescriptor accdesc = NewAccDesc(AccessType_GPR);

    accdesc.el                = if !privileged then EL0 else PSTATE.EL;
    accdesc.nontemporal       = nontemporal;
    accdesc.write              = TRUE;
    accdesc.pan                = TRUE;
    accdesc.mops               = TRUE;
    accdesc.tagaccess          = TRUE;
    accdesc.transactional      = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVE

```
// CreateAccDescSVE()
// =====
// Access descriptor for general SVE loads/stores

AccessDescriptor CreateAccDescSVE(MemOp memop, boolean nontemporal, boolean contiguous,
    boolean tagchecked)
    AccessDescriptor accdesc = 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 && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked    = FALSE;
    else
        accdesc.tagchecked    = tagchecked;
    accdesc.transactional      = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVEFF

```
// CreateAccDescSVEFF()
// =====
// Access descriptor for first-fault SVE loads

AccessDescriptor CreateAccDescSVEFF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType_SVE);

    accdesc.read               = TRUE;
    accdesc.pan                = TRUE;
    accdesc.firstfault         = TRUE;
    accdesc.first               = TRUE;
    accdesc.contiguous         = contiguous;
    accdesc.streamingsve       = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked    = FALSE;
    else
        accdesc.tagchecked    = tagchecked;
    accdesc.transactional      = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescSVENF

```
// CreateAccDescSVENF()
// =====
// Access descriptor for non-fault SVE loads

AccessDescriptor CreateAccDescSVENF(boolean contiguous, boolean tagchecked)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_SVE);

    accdesc.read          = TRUE;
    accdesc.pan           = TRUE;
    accdesc.nonfault      = TRUE;
    accdesc.contiguous    = contiguous;
    accdesc.streamingsve  = InStreamingMode();
    if (accdesc.streamingsve && boolean IMPLEMENTATION_DEFINED
        "No tag checking of SIMD&FP loads and stores in Streaming SVE mode") then
        accdesc.tagchecked = FALSE;
    else
        accdesc.tagchecked = tagchecked;
    accdesc.transactional = IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescTRBE

```
// CreateAccDescTRBE()
// =====
// Access descriptor for memory accesses by Trace Buffer Unit

AccessDescriptor CreateAccDescTRBE(SecurityState owning_ss, bits(2) owning_el)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_TRBE);

    accdesc.el           = owning_el;
    accdesc.ss           = owning_ss;
    accdesc.write        = TRUE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/CreateAccDescTTEUpdate

```
// CreateAccDescTTEUpdate()
// =====
// Access descriptor for translation table entry HW update

AccessDescriptor CreateAccDescTTEUpdate(AccessDescriptor accdesc_in)
    AccessDescriptor accdesc = NewAccDesc(AccessType\_TTW);

    accdesc.el           = accdesc_in.el;
    accdesc.ss           = accdesc_in.ss;
    accdesc.atomicop     = TRUE;
    accdesc.modop        = MemAtomicOp\_CAS;
    accdesc.read         = TRUE;
    accdesc.write        = TRUE;
    accdesc.mpam         = accdesc_in.mpam;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
// DataMemoryBarrier()
// =====

DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

## Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
// DataSynchronizationBarrier()
// =====

DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

## Library pseudocode for shared/functions/memory/DeviceType

```
// DeviceType
// =====
// Extended memory types for Device memory.

enumeration DeviceType {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".

bit EffectiveMTX(bits(64) address, boolean is_instr, bits(2) el)
    bit mtx;
    assert HaveEL(el);
    regime = S1TranslationRegime(el);
    assert(!ELUsingAArch32(regime));

    if !IsFeatureImplemented(FEAT_MTE4) || 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;
            when EL3
                mtx = TCR_EL3.MTX;

    return mtx;
```

## Library pseudocode for shared/functions/memory/EffectiveTBI

```
// EffectiveTBI()
// =====
// Returns the effective TBI in the AArch64 stage 1 translation regime for "el".

bit EffectiveTBI(bits(64) address, boolean IsInstr, bits(2) el)
    bit tbi;
    bit tbid;
    assert HaveEL(el);
    regime = 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;
        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;
            else
                tbi = TCR_EL2.TBI;
                if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if IsFeatureImplemented(FEAT_PAuth) then tbid = TCR_EL3.TBID;

    return (if (tbi == '1' && (!IsFeatureImplemented(FEAT_PAuth) || tbid == '0' ||
        !IsInstr)) then '1' else '0');
```

## Library pseudocode for shared/functions/memory/EffectiveTCMA

```
// EffectiveTCMA()
// =====
// Returns the effective TCMA of a virtual address in the stage 1 translation regime for "el".

bit EffectiveTCMA(bits(64) address, bits(2) el)
    bit tcma;
    assert HaveEL(el);
    regime = 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;
        when EL3
            tcma = TCR_EL3.TCMA;

    return tcma;
```

## Library pseudocode for shared/functions/memory/ErrorState

```
// ErrorState
// =====
// The allowed error states that can be returned by memory and used by the PE.

enumeration ErrorState {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.

enumeration Fault {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_HWUpdateAccessFlag,
                  Fault_Lockdown,
                  Fault_Exclusive,
                  Fault_ICacheMaint};
```

## Library pseudocode for shared/functions/memory/FaultRecord

```
// FaultRecord
// =====
// Fields that relate only to Faults.

type FaultRecord is (
    Fault          statuscode,          // Fault Status
    AccessDescriptor accessdesc,      // Details of the faulting access
    bits(64)      vaddress,           // Faulting virtual address
    FullAddress    ipaddress,          // Intermediate physical address
    GPCFRecord     gpcf,               // Granule Protection Check Fault record
    FullAddress    paddress,           // Physical address
    boolean       gpcfs2walk,         // GPC for a stage 2 translation table walk
    boolean       s2fslwalk,         // Is on a Stage 1 translation table walk
    boolean       write,              // TRUE for a write, FALSE for a read
    boolean       sltagnotdata,       // TRUE for a fault due to tag not accessible at stage 1.
    boolean       tagaccess,          // TRUE for a fault due to NoTagAccess permission.
    integer       level,              // For translation, access flag and Permission faults
    bit           extflag,            // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean       secondstage,        // Is a Stage 2 abort
    boolean       assuredonly,        // Stage 2 Permission fault due to AssuredOnly attribute
    boolean       toplevel,           // Stage 2 Permission fault due to TopLevel
    boolean       overlay,            // Fault due to overlay permissions
    boolean       dirtybit,           // Fault due to dirty state
    bits(4)       domain,             // Domain number, AArch32 only
    ErrorState     merrorstate,        // Incoming error state from memory
    boolean       hdbssf,             // Fault caused by HDBSS
    WatchpointInfo watchptinfo,       // Watchpoint related fields
    bits(4)       debugmoe           // 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 56 bits to allow interprocessing with AArch64.
// The maximum physical or intermediate physical address size is IMPLEMENTATION DEFINED,
// but never exceeds 56 bits.

type FullAddress is (
    PASpace    paspace,
    bits(56)  address
)
```

## Library pseudocode for shared/functions/memory/GPCF

```
// GPCF
// ====
// Possible Granule Protection Check Fault reasons

enumeration GPCF {
    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 is (
    GPCF    gpfc,
    integer level
)
```

## 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.

Hint_Prefetch(bits(64) address, PrefetchHint hint, integer target, boolean stream);
```

## 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.

Hint_RangePrefetch(bits(64) address, integer length, integer stride,
    integer count, integer reuse, bits(6) operation);
```

## Library pseudocode for shared/functions/memory/IsContiguousSVEAccess

```
// IsContiguousSVEAccess()
// =====
// Return TRUE if memory access is contiguous load/stores in an SVE mode.

boolean IsContiguousSVEAccess(AccessDescriptor accdesc)
    return (IsFeatureImplemented(FEAT_SVE) &&
        accdesc.acctype == AccessType\_SVE &&
        accdesc.contiguous);
```

## Library pseudocode for shared/functions/memory/IsDataAccess

```
// IsDataAccess()
// =====
// Return TRUE if access is to data memory.

boolean IsDataAccess(AccessType acctype)
    return ! acctype IN {AccessType\_IFETCH,
        AccessType\_TTW,
        AccessType\_DC,
        AccessType\_IC,
        AccessType\_AT};
```

## 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

boolean IsRelaxedWatchpointAccess(AccessDescriptor accdesc)
    return (IsContiguousSVEAccess(accdesc) ||
           IsSMEAccess(accdesc) ||
           (IsSIMDFPAccess(accdesc) && InStreamingMode()));
```

## Library pseudocode for shared/functions/memory/IsSIMDFPAccess

```
// IsSIMDFPAccess()
// =====
// Return TRUE if access is SIMD&FP.

boolean IsSIMDFPAccess(AccessDescriptor accdesc)
    return accdesc.acctype == AccessType\_ASIMD;
```

## Library pseudocode for shared/functions/memory/IsSMEAccess

```
// IsSMEAccess()
// =====
// Return TRUE if access is of SME load/stores.

boolean IsSMEAccess(AccessDescriptor accdesc)
    return IsFeatureImplemented(FEAT_SME) && accdesc.acctype == AccessType\_SME;
```

## Library pseudocode for shared/functions/memory/MBReqDomain

```
// MBReqDomain
// =====
// Memory barrier domain.

enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,
                           MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

## Library pseudocode for shared/functions/memory/MBReqTypes

```
// MBReqTypes
// =====
// Memory barrier read/write.

enumeration MBReqTypes     {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

## Library pseudocode for shared/functions/memory/MemAtomicOp

```
// MemAtomicOp
// =====
// Atomic data processing instruction types.

enumeration MemAtomicOp {
    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 is (
    bits(2) attrs, // See MemAttr_*, Cacheability attributes
    bits(2) hints, // See MemHint_*, Allocation hints
    boolean transient
)
```

## Library pseudocode for shared/functions/memory/MemOp

```
// MemOp
// =====
// Memory access instruction types.

enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

## Library pseudocode for shared/functions/memory/MemType

```
// MemType
// =====
// Basic memory types.

enumeration MemType {MemType_Normal, MemType_Device};
```

## Library pseudocode for shared/functions/memory/Memory

```
// Memory Tag type
// =====

enumeration MemTagType {
    MemTag_Untagged,
    MemTag_AllocationTagged,
    MemTag_CanonicallyTagged
};
```

## Library pseudocode for shared/functions/memory/MemoryAttributes

```
// MemoryAttributes
// =====
// Memory attributes descriptor

type MemoryAttributes is (
    MemType      memtype,
    DeviceType   device,          // For Device memory types
    MemAttrHints inner,          // Inner hints and attributes
    MemAttrHints outer,         // Outer hints and attributes
    Shareability shareability,  // Shareability attribute
    MemTagType   tags,          // MTE tag type for this memory.
    boolean      notagaccess,   // Allocation Tag access permission
    bit          xs             // XS attribute
);
```

## Library pseudocode for shared/functions/memory/NewAccDesc

```
// NewAccDesc()
// =====
// Create a new AccessDescriptor with initialised fields

AccessDescriptor NewAccDesc(AccessType acctype)
    AccessDescriptor accdesc;

    accdesc.acctype          = acctype;
    accdesc.el               = PSTATE.EL;
    accdesc.ss               = SecurityStateAtEL(PSTATE.EL);
    accdesc.acqsc            = FALSE;
    accdesc.acqpc            = FALSE;
    accdesc.relsc            = 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.streamingsve     = FALSE;
    accdesc.ls64             = FALSE;
    accdesc.withstatus       = FALSE;
    accdesc.mops             = FALSE;
    accdesc.a32lsmd          = FALSE;
    accdesc.tagchecked        = FALSE;
    accdesc.tagaccess        = FALSE;
    accdesc.stzgm            = FALSE;
    accdesc.transactional    = FALSE;
    accdesc.mpam             = GenMPAMCurEL(acctype);
    accdesc.ispair           = FALSE;
    accdesc.highestaddressfirst = FALSE;

    return accdesc;
```

## Library pseudocode for shared/functions/memory/PASpace

```
// PASpace
// =====
// Physical address spaces

enumeration PASpace {
    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 is (
    bits(2) ap_table,    // Stage 1 hierarchical access permissions
    bit     xn_table,    // Stage 1 hierarchical execute-never for single EL regimes
    bit     pxn_table,   // Stage 1 hierarchical privileged execute-never
    bit     uxnx_table,  // Stage 1 hierarchical unprivileged execute-never
    bits(3) ap,          // Stage 1 access permissions
    bit     xn,          // Stage 1 execute-never for single EL regimes
    bit     uxnx,        // Stage 1 unprivileged execute-never
    bit     pxnx,        // Stage 1 privileged execute-never
    bits(4) ppi,         // Stage 1 privileged indirect permissions
    bits(4) upi,         // Stage 1 unprivileged indirect permissions
    bit     ndirty,      // Stage 1 dirty state for indirect permissions scheme
    bits(4) s2pi,        // Stage 2 indirect permissions
    bit     s2dirty,     // Stage 2 dirty state
    bits(4) po_index,    // Stage 1 overlay permissions index
    bits(4) s2po_index,  // Stage 2 overlay permissions index
    bits(2) s2ap,        // Stage 2 access permissions
    bit     s2tag_na,    // Stage 2 tag access
    bit     s2xnx,       // Stage 2 extended execute-never
    bit     dbm,         // Dirty bit management
    bit     s2xn         // 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.

(PhysMemRetStatus, bits(8*size)) PhysMemRead(AddressDescriptor desc, integer size,
AccessDescriptor accdesc);
```

## Library pseudocode for shared/functions/memory/PhysMemRetStatus

```
// PhysMemRetStatus
// =====
// Fields that relate only to return values of PhysMem functions.

type PhysMemRetStatus is (
    Fault      statuscode,    // Fault Status
    bit        extflag,      // IMPLEMENTATION DEFINED syndrome for External aborts
    ErrorState merrorstate,  // Optional error state returned on a physical memory access
    bits(64)   store64bstatus // 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.

PhysMemRetStatus PhysMemWrite(AddressDescriptor desc, integer size, AccessDescriptor accdesc,
    bits(8*size) value);
```

## Library pseudocode for shared/functions/memory/PrefetchHint

```
// PrefetchHint
// =====
// Prefetch hint types.

enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

## Library pseudocode for shared/functions/memory/S1AccessControls

```
// S1AccessControls
// =====
// Effective access controls defined by stage 1 translation

type S1AccessControls is (
    bit r,                // Stage 1 base read permission
    bit w,                // Stage 1 base write permission
    bit x,                // Stage 1 base execute permission
    bit gcs,              // Stage 1 GCS permission
    boolean overlay,      // Stage 1 overlay feature enabled
    bit or,               // Stage 1 overlay read permission
    bit ow,               // Stage 1 overlay write permission
    bit ox,               // Stage 1 overlay execute permission
    bit wxn               // 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 is (
    bit r,                // Stage 2 read permission.
    bit w,                // Stage 2 write permission.
    bit x,                // Stage 2 execute permission.
    bit r_rcw,            // Stage 2 Read perms for RCW instruction.
    bit w_rcw,            // Stage 2 Write perms for RCW instruction.
    bit r_mmu,            // Stage 2 Read perms for TTW data.
    bit w_mmu,            // Stage 2 Write perms for TTW data.
    bit toplevel0,        // IPA as top level table for TTBR0_EL1.
    bit toplevel1,        // IPA as top level table for TTBR1_EL1.
    boolean overlay,      // Overlay enable
    bit or,               // Stage 2 overlay read permission.
    bit ow,               // Stage 2 overlay write permission.
    bit ox,               // Stage 2 overlay execute permission.
    bit or_rcw,           // Stage 2 overlay Read perms for RCW instruction.
    bit ow_rcw,           // Stage 2 overlay Write perms for RCW instruction.
    bit or_mmu,           // Stage 2 overlay Read perms for TTW data.
    bit ow_mmu,           // Stage 2 overlay Write perms for TTW data.
)
```

## Library pseudocode for shared/functions/memory/Shareability

```
// Shareability
// =====

enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

### Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
// SpeculativeStoreBypassBarrierToPA()
// =====

SpeculativeStoreBypassBarrierToPA();
```

### Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
// SpeculativeStoreBypassBarrierToVA()
// =====

SpeculativeStoreBypassBarrierToVA();
```

### Library pseudocode for shared/functions/memory/Tag

```
// Tag Granule size
// =====

constant integer LOG2_TAG_GRANULE = 4;

constant integer TAG_GRANULE = 1 << LOG2\_TAG\_GRANULE;
```

### Library pseudocode for shared/functions/memory/VARange

```
// VARange
// =====
// Virtual address ranges

enumeration VARange {
    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.

PARTIDSpaceType AltPARTIDSpace(bits(2) el, SecurityState security,
                                PARTIDSpaceType primaryPIDSpace)
    case security of
        when SS\_NonSecure
            assert el != EL3;
            return primaryPIDSpace;
        when SS\_Secure
            assert el != EL3;
            if primaryPIDSpace == PIDSpace\_NonSecure then
                return primaryPIDSpace;
            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;
            else
                return primaryPIDSpace;
        when SS\_Realm
            assert el != EL3;
            return AltPIDRealm(el, primaryPIDSpace);
        otherwise
            Unreachable();
```

## 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.

PARTIDSpaceType AltPIDRealm(bits(2) el, PARTIDSpaceType primaryPIDSpace)
    PARTIDSpaceType PIDSpace = primaryPIDSpace;
    case el of
        when EL0
            if ELIsInHost(EL0) then
                if !UsePrimarySpaceEL2() then
                    PIDSpace = PIDSpace\_NonSecure;
            elsif !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace\_NonSecure;
        when EL1
            if !UsePrimarySpaceEL10() then
                PIDSpace = PIDSpace\_NonSecure;
        when EL2
            if !UsePrimarySpaceEL2() then
                PIDSpace = PIDSpace\_NonSecure;
        otherwise
            Unreachable();
    return PIDSpace;
```

## 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.

PARTIDSpaceType AltPIDSecure(bits(2) el, PARTIDSpaceType primaryPIDSpace)
PARTIDSpaceType PIDSpace = primaryPIDSpace;
case el of
  when EL0
    if EL2Enabled() then
      if ELIsInHost(EL0) then
        if !UsePrimarySpaceEL2() then
          PIDSpace = PIDSpace NonSecure;
        elsif !UsePrimarySpaceEL10() then
          PIDSpace = PIDSpace NonSecure;
      elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
        PIDSpace = PIDSpace NonSecure;
  when EL1
    if EL2Enabled() then
      if !UsePrimarySpaceEL10() then
        PIDSpace = PIDSpace NonSecure;
      elsif MPAM3_EL3.ALTSP_HEN == '0' && MPAM3_EL3.ALTSP_HFC == '1' then
        PIDSpace = PIDSpace NonSecure;
  when EL2
    if !UsePrimarySpaceEL2() then
      PIDSpace = PIDSpace NonSecure;
  otherwise
    Unreachable();
return PIDSpace;
```

## 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.

MPAMInfo DefaultMPAMInfo(PARTIDSpaceType partidspace)
MPAMInfo defaultinfo;
defaultinfo.mpam_sp = partidspace;
defaultinfo.partid = DEFAULT\_PARTID;
defaultinfo.pmg = DEFAULT\_PMG;
return defaultinfo;
```

## Library pseudocode for shared/functions/mpam/GenMPAM

```
// GenMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If in_d is TRUE returns MPAM information using PARTID_I and PMG_I fields
// of MPAMel_ELx register and otherwise using PARTID_D and PMG_D fields.
// If in_sm is TRUE returns MPAM information using PARTID_D and PMG_D fields
// of MPAMSM_EL1 register.
// Produces a PARTID in PARTID space pspace.

MPAMinfo GenMPAM(bits(2) el, boolean in_d, boolean in_sm, PARTIDSpaceType pspace)
    MPAMinfo returninfo;
    PARTIDType partidel;
    boolean perr;
    // gstplk is guest OS application locked by the EL2 hypervisor to
    // only use EL1 the virtual machine's PARTIDs.
    constant boolean gstplk = (el == EL0 && EL2Enabled() &&
        MPAMHCR_EL2.GSTAPP_PLK == '1' &&
        HCR_EL2.TGE == '0');
    constant bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = GenPARTID(eff_el, in_d, in_sm);
    constant PMGType groupel = GenPMG(eff_el, in_d, in_sm, perr);
    returninfo.mpam_sp = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
```

## 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.

MPAMInfo GenMPAMAtEL(AccessType acctype, bits(2) el)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    constant SecurityState security = SecurityStateAtEL(el);
    boolean in_d = FALSE;
    boolean in_sm = FALSE;
    PARTIDSpaceType pspace = PARTIDSpaceFromSS(security);
    if pspace == PIDSpace NonSecure && !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    if UsingAArch32() then
        (validEL, mpamEL) = ELFromM32(PSTATE.M);
    else
        mpamEL = if acctype == AccessType NV2 then EL2 else el;
        validEL = TRUE;
    case acctype of
        when AccessType IFETCH, AccessType IC
            in_d = TRUE;
        when AccessType SME
            in_sm = (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                    boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence");
        when AccessType FP, AccessType ASIMD, AccessType SVE
            in_sm = (IsFeatureImplemented(FEAT_SME) && PSTATE.SM == '1' &&
                    (boolean IMPLEMENTATION_DEFINED "Shared SMCU" ||
                     boolean IMPLEMENTATION_DEFINED "MPAMSM_EL1 label precedence"));
        otherwise
            // Other access types are DATA accesses
            in_d = FALSE;
    if !validEL then
        return DefaultMPAMInfo(pspace);
    elsif IsFeatureImplemented(FEAT_RME) && MPAMIDR_EL1.HAS_ALTSP == '1' then
        // Substitute alternative PARTID space if selected
        pspace = AltPARTIDSpace(mpamEL, security, pspace);
    if IsFeatureImplemented(FEAT_MPAMv0p1) && MPAMIDR_EL1.HAS_FORCE_NS == '1' then
        if MPAM3_EL3.FORCE_NS == '1' && security == SS Secure then
            pspace = PIDSpace NonSecure;
    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);
    if !MPAMIsEnabled() then
        return DefaultMPAMInfo(pspace);
    else
        return GenMPAM(mpamEL, in_d, in_sm, pspace);
```

## 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.

MPAMInfo GenMPAMCurEL(AccessType acctype)
    return GenMPAMAtEL(acctype, PSTATE.EL);
```

## Library pseudocode for shared/functions/mpam/GenPARTID

```
// GenPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If in_d is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.
// If in_sm is TRUE then PARTID is from MPAMSM_EL1.PARTID_D.

(PARTIDType, boolean) GenPARTID(bits(2) el, boolean in_d, boolean in_sm)
    constant PARTIDType partidel = GetMPAM_PARTID(el, in_d, in_sm);
    constant PARTIDType partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DEFAULT_PARTID, TRUE);
    if MPAMIsVirtual(el) then
        return MAP_vPARTID(partidel);
    else
        return (partidel, FALSE);
```

## Library pseudocode for shared/functions/mpam/GenPMG

```
// GenPMG()
// =====
// Returns PMG for exception level el and I- or D-side (in_d).
// If PARTID generation (GenPARTID) encountered an error, GenPMG() should be
// called with partid_err as TRUE.

PMGType GenPMG(bits(2) el, boolean in_d, boolean in_sm, boolean partid_err)
    constant integer pmg_max = UInt(MPAMIDR_EL1.PMG_MAX);
    // It is CONSTRAINED UNPREDICTABLE whether partid_err forces PMG to
    // use the default or if it uses the PMG from getMPAM_PMG.
    if partid_err then
        return DEFAULT_PMG;
    constant PMGType groupel = GetMPAM_PMG(el, in_d, in_sm);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DEFAULT_PMG;
```

## Library pseudocode for shared/functions/mpam/GetMPAM\_PARTID

```
// GetMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If in_sm is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If in_d is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDType GetMPAM_PARTID(bits(2) MPAMn, boolean in_d, boolean in_sm)
    PARTIDType partid;

    if in_sm then
        partid = MPAMSM_EL1.PARTID_D;
        return partid;

    if in_d then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if EL2Enabled() then MPAM2_EL2.PARTID_I else DEFAULT_PARTID;
            when '01' partid = MPAM1_EL1.PARTID_I;
            when '00' partid = MPAM0_EL1.PARTID_I;
            otherwise partid = PARTIDType UNKNOWN;
        else
            case MPAMn of
                when '11' partid = MPAM3_EL3.PARTID_D;
                when '10' partid = if EL2Enabled() then MPAM2_EL2.PARTID_D else DEFAULT_PARTID;
                when '01' partid = MPAM1_EL1.PARTID_D;
                when '00' partid = MPAM0_EL1.PARTID_D;
                otherwise partid = PARTIDType UNKNOWN;
            return partid;
```

## Library pseudocode for shared/functions/mpam/GetMPAM\_PMG

```
// GetMPAM_PMG()
// =====
// Returns a PMG from one of the MPAMn_ELx or MPAMSM_EL1 registers.
// If in_sm is TRUE, the MPAMSM_EL1 register is used. Otherwise,
// MPAMn selects the MPAMn_ELx register used.
// If in_d is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGType GetMPAM_PMG(bits(2) MPAMn, boolean in_d, boolean in_sm)
    PMGType pmg;

    if in_sm then
        pmg = MPAMSM_EL1.PMG_D;
        return pmg;

    if in_d then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if EL2Enabled() then MPAM2_EL2.PMG_I else DEFAULT_PMG;
            when '01' pmg = MPAM1_EL1.PMG_I;
            when '00' pmg = MPAM0_EL1.PMG_I;
            otherwise pmg = PMGType UNKNOWN;
        else
            case MPAMn of
                when '11' pmg = MPAM3_EL3.PMG_D;
                when '10' pmg = if EL2Enabled() then MPAM2_EL2.PMG_D else DEFAULT_PMG;
                when '01' pmg = MPAM1_EL1.PMG_D;
                when '00' pmg = MPAM0_EL1.PMG_D;
                otherwise pmg = PMGType UNKNOWN;
            return pmg;
```

## 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.

(PARTIDType, boolean) MAP_vPARTID(PARTIDType vpartid)
    // should not ever be called if EL2 is not implemented
    // or is implemented but not enabled in the current
    // security state.
    PARTIDType ret;
    boolean err;
    integer virt      = UInt(vpartid);
    constant integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    constant integer vpartid_max = (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);

    // Check for valid mapping entry.
    if MPAMVPMV_EL2<virt> == '1' then
        // vpartid has a valid mapping so access the map.
        ret = mapvpmw(virt);
        err = FALSE;

    // Is the default virtual PARTID valid?
    elseif MPAMVPMV_EL2<0> == '1' then
        // Yes, so use default mapping for vpartid == 0.
        ret = MPAMVPM0_EL2<0 +: 16>;
        err = FALSE;

    // Neither is valid so use default physical PARTID.
    else
        ret = DEFAULT_PARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    constant integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DEFAULT_PARTID;
        err = TRUE;
    return (ret, err);
```

## Library pseudocode for shared/functions/mpam/MPAM

```
constant PARTIDType DEFAULT_PARTID = 0<15:0>;
constant PMGType     DEFAULT_PMG    = 0<7:0>;

// Defines the MPAM_engine_. The _engine_ produces the MPAM labels for memory
// accesses from the state information stored in the MPAM System registers.

// The MPAM_engine_ runs in all states and with the MPAM AArch64 system
// registers and PE execution state controlling its behavior.

// MPAM Types
// =====

type PARTIDType = bits(16);

type PMGType = bits(8);

enumeration PARTIDSpaceType {
    PIDSpace_Secure,
    PIDSpace_Root,
    PIDSpace_Realm,
    PIDSpace_NonSecure
};

type MPAMInfo is (
    PARTIDSpaceType mpam_sp,
    PARTIDType partid,
    PMGType pmg
)
```

## Library pseudocode for shared/functions/mpam/MPAMIsEnabled

```
// MPAMIsEnabled()
// =====
// Returns TRUE if MPAMIsEnabled.

boolean MPAMIsEnabled()
    el = 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';
```

## Library pseudocode for shared/functions/mpam/MPAMIsVirtual

```
// MPAMIsVirtual()
// =====
// Returns TRUE if MPAM is configured to be virtual at EL.

boolean MPAMIsVirtual(bits(2) el)
    return (MPAMIDR_EL1.HAS_HCR == '1' && EL2Enabled() &&
        ((el == EL0 && MPAMHCR_EL2.EL0_VPMEN == '1' && !ELIsInHost(EL0)) ||
        (el == EL1 && MPAMHCR_EL2.EL1_VPMEN == '1')));
```



## Library pseudocode for shared/functions/mpam/PARTIDSpaceFromSS

```
// PARTIDSpaceFromSS()
// =====
// Returns the primary PARTID space from the Security State.

PARTIDSpaceType PARTIDSpaceFromSS(SecurityState security)
    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();
```

## 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.

boolean UsePrimarySpaceEL10()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMIsEnabled() || !EL2Enabled() || MPAM2_EL2.ALTSP_HFC == '0';
```

## 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.

boolean UsePrimarySpaceEL2()
    if MPAM3_EL3.ALTSP_HEN == '0' then
        return MPAM3_EL3.ALTSP_HFC == '0';
    return !MPAMIsEnabled() || MPAM2_EL2.ALTSP_EL2 == '0';
```

## 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.

PARTIDType mapvpmw(integer vpartid)
    bits(64) vpmw;
    constant integer wd = vpartid DIV 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);
    // vpme_lsb selects LSB of field within register
    constant integer vpme_lsb = (vpartid MOD 4) * 16;
    return vpmw<vpme_lsb +: 16>;
```

## Library pseudocode for shared/functions/predictionrestrict/ASID

```
// ASID[]
// =====
// Effective ASID.

bits(16) ASID[]
    if ELIsInHost(EL0) then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;
    if !ELUsingAArch32(EL1) then
        if TCR_EL1.A1 == '1' then
            return TTBR1_EL1.ASID;
        else
            return TTBR0_EL1.ASID;
    else
        if TTBCR.EAE == '0' then
            return ZeroExtend(CONTEXTIDR.ASID, 16);
        else
            if TTBCR.A1 == '1' then
                return ZeroExtend(TTBR1.ASID, 16);
            else
                return ZeroExtend(TTBR0.ASID, 16);
```

## Library pseudocode for shared/functions/predictionrestrict/ExecutionCntxt

```
// ExecutionCntxt
// =====
// Context information for prediction restriction operation.

type ExecutionCntxt is (
    boolean        is_vmid_valid, // is vmid valid for current context
    boolean        all_vmid,      // should the operation be applied for all vmids
    bits(16)       vmid,          // if all_vmid = FALSE, vmid to which operation is applied
    boolean        is_asid_valid, // is asid valid for current context
    boolean        all_asid,      // should the operation be applied for all asids
    bits(16)       asid,          // if all_asid = FALSE, ASID to which operation is applied
    bits(2)        target_el,     // target EL at which operation is performed
    SecurityState security,
    RestrictType  restriction    // type of restriction operation
)
```

## Library pseudocode for shared/functions/predictionrestrict/RESTRICT\_PREDICTIONS

```
// RESTRICT_PREDICTIONS()
// =====
// Clear all speculated values.

RESTRICT_PREDICTIONS(ExecutionCntxt c)
    IMPLEMENTATION_DEFINED;
```

## Library pseudocode for shared/functions/predictionrestrict/RestrictType

```
// RestrictType
// =====
// Type of restriction on speculation.

enumeration RestrictType {
    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.

SecurityState TargetSecurityState(bit NS, bit NSE)
    curr_ss = 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;
        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;
                elsif curr_ss == SS Realm then
                    return SS Realm;
            unreachable();
```

## 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.

BranchTo(bits(N) target, BranchType branch_type, boolean branch_conditional)
    Hint\_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        constant bits(64) target_vaddress = 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);
            constant boolean branch_taken = TRUE;

            if IsFeatureImplemented(FEAT_SPE) then
                SPEBranch(target, branch_type, branch_conditional, branch_taken);

        _PC = target_vaddress;
    return;
```

## 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.

BranchToAddr(bits(N) target, BranchType branch_type)
    Hint\_Branch(branch_type);
    if N == 32 then
        assert UsingAArch32();
        _PC = ZeroExtend(target, 64);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

## Library pseudocode for shared/functions/registers/BranchType

```
// BranchType
// =====
// Information associated with a change in control flow.

enumeration BranchType {
    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_TMFAIL,     // Transaction failure
    BranchType_RESET,      // Reset
    BranchType_UNKNOWN};   // Other
```

## Library pseudocode for shared/functions/registers/EffectiveFPCR

```
// EffectiveFPCR()
// =====
// Returns the effective FPCR value

FPCR_Type EffectiveFPCR()
    if UsingAArch32() then
        FPCR\_Type fpcr = ZeroExtend(FPSCR, 64);
        fpcr<7:0> = '00000000';
        fpcr<31:27> = '00000';
        return fpcr;
    return FPCR;
```

## Library pseudocode for shared/functions/registers/FPCR\_Type

```
// FPCR_Type
// =====
// A type representing the FPCR register

type FPCR_Type;
```

## Library pseudocode for shared/functions/registers/FPMR\_Type

```
// FPMR_Type
// =====
// A type representing the FPMR register

type FPMR_Type;
```

## 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.

Hint_Branch(BranchType hint);
```

## Library pseudocode for shared/functions/registers/NextInstrAddr

```
// NextInstrAddr()
// =====
// Return address of the sequentially next instruction.

bits(N) NextInstrAddr(integer N);
```

## Library pseudocode for shared/functions/registers/ResetExternalDebugRegisters

```
// ResetExternalDebugRegisters()
// =====
// Reset the External Debug registers in the Core power domain.

ResetExternalDebugRegisters(boolean cold_reset);
```

## Library pseudocode for shared/functions/registers/ThisInstrAddr

```
// ThisInstrAddr()
// =====
// Return address of the current instruction.

bits(N) ThisInstrAddr(integer N)
    assert N == 64 || (N == 32 && UsingAArch32() );
    return _PC<N-1:0>;
```

## Library pseudocode for shared/functions/registers/UnimplementedIDRegister

```
// UnimplementedIDRegister()
// =====
// Trap access to unimplemented encodings in the feature ID register space.

UnimplementedIDRegister()
    if IsFeatureImplemented(FEAT_IDST) then
        target_el = PSTATE.EL;
        if PSTATE.EL == EL0 then
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
            AArch64.SystemAccessTrap(target_el, 0x18);
        UNDEFINED;
```

## Library pseudocode for shared/functions/registers/\_PC

```
// _PC - the program counter
// =====

bits(64) _PC;
```

## Library pseudocode for shared/functions/registers/\_R

```
// _R[] - the general-purpose register file
// =====

array bits(64) _R[0..30];
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_ELx

```
// SPSR_ELx[] - non-assignment form
// =====

bits(64) SPSR_ELx[]
    bits(64) result;
    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();
    return result;

// SPSR_ELx[] - assignment form
// =====

SPSR_ELx[] = bits(64) value
    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();
    return;
```

## Library pseudocode for shared/functions/sysregisters/SPSR\_curr

```
// SPSR_curr[] - non-assignment form
// =====

bits(32) SPSR_curr[]
  bits(32) result;
  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();
  return result;

// SPSR_curr[] - assignment form
// =====

SPSR_curr[] = bits(32) value
  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();
  return;
```

## Library pseudocode for shared/functions/system/AArch64.ChkFeat

```
// AArch64.ChkFeat()
// =====
// Indicates the status of some features

bits(64) AArch64.ChkFeat(bits(64) feat_select)
  bits(64) feat_en = Zeros(64);
  feat_en<0> = if IsFeatureImplemented(FEAT_GCS) && GCSEnabled(PSTATE.EL) then '1' else '0';
  return feat_select AND NOT(feat_en);
```

## 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.

bits(64) AddressAdd(bits(64) base, integer offset, AccessDescriptor accdesc)
  return AddressAdd(base, offset<63:0>, accdesc);

bits(64) AddressAdd(bits(64) base, bits(64) offset, AccessDescriptor accdesc)
  bits(64) result = base + offset;
  result = PointerAddCheckAtEL(accdesc.el, result, base);
  return result;
```

## 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.

bits(64) AddressIncrement(bits(64) base, integer increment, AccessDescriptor accdesc)
    return AddressIncrement(base, increment<63:0>, accdesc);

bits(64) AddressIncrement(bits(64) base, bits(64) increment, AccessDescriptor accdesc)
    bits(64) result = 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);
    return result;
```

## Library pseudocode for shared/functions/system/AddressNotInNaturallyAlignedBlock

```
// AddressNotInNaturallyAlignedBlock()
// =====
// The 'address' is not in a naturally aligned block if it doesn't meet all 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.

boolean AddressNotInNaturallyAlignedBlock(bits(64) address);
```

## 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.

BranchTargetCheck()
    assert IsFeatureImplemented(FEAT_BTI) && !UsingAArch32();

    // The branch target check considers two state variables:
    // * InGuardedPage, which is evaluated during instruction fetch.
    // * BTypeCompatible, which is evaluated during instruction decode.
    if InGuardedPage && PSTATE.BTYPE != '00' && !BTypeCompatible && !Halted() then
        constant bits(64) pc = ThisInstrAddr(64);
        AArch64.BranchTargetException(pc<51:0>);
```

## Library pseudocode for shared/functions/system/ClearEventRegister

```
// ClearEventRegister()
// =====
// Clear the Event Register of this PE.

ClearEventRegister()
    EventRegister = '0';
    return;
```



## Library pseudocode for shared/functions/system/ConditionHolds

```
// ConditionHolds()
// =====
// Return TRUE iff COND currently holds

boolean ConditionHolds(bits(4) cond)
// Evaluate base condition.
boolean result;
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

// Condition flag values in the set '111x' indicate always true
// Otherwise, invert condition if necessary.
if cond<0> == '1' && cond != '1111' then
    result = !result;

return result;
```

## Library pseudocode for shared/functions/system/ConsumptionOfSpeculativeDataBarrier

```
// ConsumptionOfSpeculativeDataBarrier()
// =====

ConsumptionOfSpeculativeDataBarrier();
```

## Library pseudocode for shared/functions/system/CurrentInstrSet

```
// CurrentInstrSet()
// =====

InstrSet CurrentInstrSet()
    InstrSet result;
    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;
    return result;
```

## Library pseudocode for shared/functions/system/CurrentPL

```
// CurrentPL()
// =====

PrivilegeLevel CurrentPL()
    return PLOfEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/CurrentSecurityState

```
// CurrentSecurityState()
// =====
// Returns the effective security state at the exception level based off current settings.

SecurityState CurrentSecurityState()
    return SecurityStateAtEL(PSTATE.EL);
```

## Library pseudocode for shared/functions/system/DSBAlias

```
// DSBAlias
// =====
// Aliases of DSB.

enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

## Library pseudocode for shared/functions/system/EL0

```
// EL0-3
// =====
// PSTATE.EL Exception level bits.

constant bits(2) EL3 = '11';
constant bits(2) EL2 = '10';
constant bits(2) EL1 = '01';
constant bits(2) EL0 = '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.

boolean EL2Enabled()
    return HaveEL\(EL2\) && (!HaveEL\(EL3\) || SCR_curr[].NS == '1' || IsSecureEL2Enabled\(\));
```

## Library pseudocode for shared/functions/system/EL3SDDUndef

```
// EL3SDDUndef()
// =====
// Returns TRUE if in Debug state and EDSCR.SDD is set.

boolean EL3SDDUndef()
    if Halted\(\) && EDSCR.SDD == '1' then
        assert (PSTATE.EL != EL3 &&
            (IsFeatureImplemented(FEAT_RME) || CurrentSecurityState\(\) != SS\_Secure));
        return TRUE;
    else
        return FALSE;
```

## 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.

boolean EL3SDDUndefPriority()
    return EL3SDDUndef\(\) && boolean IMPLEMENTATION_DEFINED "EL3 trap priority when SDD == '1'";
```

## Library pseudocode for shared/functions/system/ELFromM32

```
// ELFromM32()
// =====

(boolean, bits(2)) ELFromM32(bits(5) mode)
    // 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'.
    bits(2) el;
    boolean valid = !BadMode(mode); // Check for modes that are not valid for this implementation
    constant bits(2) effective_nse_ns = 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

    if valid && el == EL2 && HaveEL(EL3) && SCR_curr[].NS == '0' then
        valid = FALSE; // EL2 only valid in Non-secure state in AArch32

    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

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

## 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'.

(boolean, bits(2)) ELFromSPSR(bits(N) spsr)
    bits(2) el;
    boolean valid;
    if spsr<4> == '0' then // AArch64 state
        el = spsr<3:2>;
        constant bits(2) effective_nse_ns = 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() && SCR_EL3.NS == '0' then
            valid = FALSE; // Unless Secure EL2 is enabled, EL2 valid only in Non-secure state
        else
            valid = TRUE;
    elseif HaveAArch32() then // AArch32 state
        (valid, el) = ELFromM32(spsr<4:0>);
    else
        valid = FALSE;

    if !valid then el = bits(2) UNKNOWN;
    return (valid, el);
```

## Library pseudocode for shared/functions/system/ELIsInHost

```
// ELIsInHost()
// =====

boolean ELIsInHost(bits(2) el)
    if !IsFeatureImplemented(FEAT_VHE) || ELUsingAArch32(EL2) then
        return FALSE;
    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();
```

## Library pseudocode for shared/functions/system/ELStateUsingAArch32

```
// ELStateUsingAArch32()
// =====

boolean ELStateUsingAArch32(bits(2) el, boolean secure)
    // See ELStateUsingAArch32K() for description. Must only be called in circumstances where
    // result is valid (typically, that means 'el IN {EL1,EL2,EL3}').
    (known, aarch32) = ELStateUsingAArch32K(el, secure);
    assert known;
    return aarch32;
```

## 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.

(boolean, boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
    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

    // Remainder of function deals with the interprocessing cases when highest
    // Exception level is using AArch64.

    if el == EL3 then
        return (TRUE, FALSE);

    if (HaveEL(EL3) && SCR_EL3.RW == '0' &&
        (!secure || !IsFeatureImplemented(FEAT_SEL2) || SCR_EL3.EEL2 == '0')) then
        // AArch32 below EL3.
        return (TRUE, TRUE);

    if el == EL2 then
        return (TRUE, FALSE);

    if (HaveEL(EL2) && !ELIsInHost(EL0) && HCR_EL2.RW == '0' &&
        (!secure || (IsFeatureImplemented(FEAT_SEL2) && SCR_EL3.EEL2 == '1')))) then
        // AArch32 below EL2.
        return (TRUE, TRUE);

    if el == EL1 then
        return (TRUE, FALSE);

    // 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, boolean UNKNOWN);
```

## Library pseudocode for shared/functions/system/ELUsingAArch32

```
// ELUsingAArch32()
// =====

boolean ELUsingAArch32(bits(2) el)
    return ELStateUsingAArch32(el, IsSecureBelowEL3());
```

## Library pseudocode for shared/functions/system/ELUsingAArch32K

```
// ELUsingAArch32K()
// =====

(boolean,boolean) ELUsingAArch32K(bits(2) el)
    return ELStateUsingAArch32K(el, IsSecureBelowEL3());
```

## Library pseudocode for shared/functions/system/EffectiveEA

```
// EffectiveEA()
// =====
// Returns effective SCR_EL3.EA value

bit EffectiveEA()
    if !HaveEL(EL3) || (Halted() && EDSCR.SDD == '0') then
        return '0';
    else
        return if HaveAArch64() then SCR_EL3.EA else SCR.EA;
```

## Library pseudocode for shared/functions/system/EffectiveHCR\_EL2\_E2H

```
// EffectiveHCR_EL2_E2H()
// =====
// Return the Effective HCR_EL2.E2H value.

bit EffectiveHCR_EL2_E2H()
    if !IsFeatureImplemented(FEAT_VHE) then
        return '0';

    if !IsFeatureImplemented(FEAT_E2H0) then
        return '1';

    return HCR_EL2.E2H;
```

## Library pseudocode for shared/functions/system/EffectiveHCR\_EL2\_NVx

```
// EffectiveHCR_EL2_NVx()
// =====
// Return the Effective value of HCR_EL2.<NV2,NV1,NV>.

bits(3) EffectiveHCR_EL2_NVx()
    if !EL2Enabled() || !IsFeatureImplemented(FEAT_NV) then
        return '000';

    bit nv1 = HCR_EL2.NV1;
    if (!IsFeatureImplemented(FEAT_E2H0) &&
        boolean IMPLEMENTATION_DEFINED "HCR_EL2.NV1 is implemented as RAZ") then
        nv1 = '0';

    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';
            else
                return '000';

    if !IsFeatureImplemented(FEAT_NV2) then
        return '0' : nv1 : '1';

    bit nv2 = HCR_EL2.NV2;
    if (nv2 == '0' && boolean IMPLEMENTATION_DEFINED
        "Programming HCR_EL2.<NV,NV2> to '10' behaves as '11'") then
        nv2 = '1';

    return nv2 : nv1 : '1';
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NS

```
// EffectiveSCR_EL3_NS()
// =====
// Return Effective SCR_EL3.NS value.

bit EffectiveSCR_EL3_NS()
    if !HaveSecureState() then
        return '1';
    elsif !HaveEL(EL3) then
        return '0';
    elsif ELUsingAArch32(EL3) then
        return SCR.NS;
    else
        return SCR_EL3.NS;
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_NSE

```
// EffectiveSCR_EL3_NSE()
// =====
// Return Effective SCR_EL3.NSE value.

bit EffectiveSCR_EL3_NSE()
    return if !IsFeatureImplemented(FEAT_RME) then '0' else SCR_EL3.NSE;
```

### Library pseudocode for shared/functions/system/EffectiveSCR\_EL3\_RW

```
// EffectiveSCR_EL3_RW()
// =====
// Returns effective SCR_EL3.RW value

bit EffectiveSCR_EL3_RW()
    if !HaveAArch64() then
        return '0';
    if !HaveAArch32EL(EL2) && !HaveAArch32EL(EL1) then
        return '1';
    if HaveAArch32EL(EL1) then
        if !HaveAArch32EL(EL2) && SCR_EL3.NS == '1' then
            return '1';
        if IsFeatureImplemented(FEAT_SEL2) && SCR_EL3.EEL2 == '1' && SCR_EL3.NS == '0' then
            return '1';
    return SCR_EL3.RW;
```

### Library pseudocode for shared/functions/system/EffectiveTGE

```
// EffectiveTGE()
// =====
// Returns effective TGE value

bit EffectiveTGE()
    if EL2Enabled() then
        return if ELUsingAArch32(EL2) then HCR.TGE else HCR_EL2.TGE;
    else
        return '0'; // Effective value of TGE is zero
```

### Library pseudocode for shared/functions/system/EndOfInstruction

```
// EndOfInstruction()
// =====
// Terminate processing of the current instruction.

EndOfInstruction();
```

### Library pseudocode for shared/functions/system/EnterLowPowerState

```
// EnterLowPowerState()
// =====
// PE enters a low-power state.

EnterLowPowerState();
```

### Library pseudocode for shared/functions/system/EventRegister

```
// EventRegister
// =====
// Event Register for this PE.

bits(1) EventRegister;
```

### 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.

enumeration ExceptionalOccurrenceTargetState {
    AArch32_NonDebugState,
    AArch64_NonDebugState,
    DebugState
};
```

### Library pseudocode for shared/functions/system/ExecuteAsNOP

```
// ExecuteAsNOP()
// =====

ExecuteAsNOP()
    EndOfInstruction();
```

### 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.

(boolean, boolean) FIQPending();
```

### Library pseudocode for shared/functions/system/GetAccumulatedFPExceptions

```
// GetAccumulatedFPExceptions()
// =====
// Returns FP exceptions accumulated by the PE.

bits(8) GetAccumulatedFPExceptions();
```

### 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.

bits(2) GetLoadStoreType();
```



## Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState, integer N)
    if UsingAArch32() && targetELState == AArch32\_NonDebugState then
        assert N == 32;
    else
        assert N == 64;

    bits(N) spsr = Zeros(N);
    if IsFeatureImplemented(FEAT_UINJ) then
        spsr<36> = if targetELState == DebugState then PSTATE.UINJ else '0';
    spsr<31:28> = PSTATE.<N,Z,C,V>;
    if IsFeatureImplemented(FEAT_PAN) then spsr<22> = PSTATE.PAN;
    spsr<20> = PSTATE.IL;
    if PSTATE.nRW == '1' then // AArch32 state
        if IsFeatureImplemented(FEAT_SEBEP) && targetELState != AArch32\_NonDebugState then
            spsr<33> = PSTATE.PPEND;
        spsr<27> = PSTATE.Q;
        spsr<26:25> = PSTATE.IT<1:0>;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<23> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_DIT) then
            if targetELState == AArch32\_NonDebugState then
                spsr<21> = PSTATE.DIT;
            else // AArch64_NonDebugState or DebugState
                spsr<24> = PSTATE.DIT;
        if targetELState IN {AArch64\_NonDebugState, DebugState} then
            spsr<21> = PSTATE.SS;
        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;
        if IsFeatureImplemented(FEAT_GCS) then spsr<34> = PSTATE.EXLOCK;
        if IsFeatureImplemented(FEAT_SEBEP) then spsr<33> = PSTATE.PPEND;
        if (IsFeatureImplemented(FEAT_EBEP) || IsFeatureImplemented(FEAT_SPE_EXC) ||
            IsFeatureImplemented(FEAT_TRBE_EXC)) then
            spsr<32> = PSTATE.PM;
        if IsFeatureImplemented(FEAT_MTE) then spsr<25> = PSTATE.TCO;
        if IsFeatureImplemented(FEAT_DIT) then spsr<24> = PSTATE.DIT;
        if IsFeatureImplemented(FEAT_UAO) then spsr<23> = PSTATE.UAO;
        spsr<21> = PSTATE.SS;
        if IsFeatureImplemented(FEAT_NMI) then spsr<13> = PSTATE.ALLINT;
        if IsFeatureImplemented(FEAT_SSBS) then spsr<12> = PSTATE.SSBS;
        if IsFeatureImplemented(FEAT_BTI) then spsr<11:10> = PSTATE.BTYPE;
        spsr<9:6> = PSTATE.<D,A,I,F>;
        spsr<4> = PSTATE.nRW;
        spsr<3:2> = PSTATE.EL;
        spsr<0> = PSTATE.SP;
    return spsr;
```

## Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
    return IsFeatureImplemented(FEAT_AA32EL0);
```

### Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// =====
// Return TRUE if Exception level 'el' supports AArch32 in this implementation

boolean HaveAArch32EL(bits(2) el)
    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);
```

### Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return (IsFeatureImplemented(FEAT_AA64EL0)
        || IsFeatureImplemented(FEAT_AA64EL1)
        || IsFeatureImplemented(FEAT_AA64EL2)
        || IsFeatureImplemented(FEAT_AA64EL3)
    );
```

### Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    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();
```

### Library pseudocode for shared/functions/system/HaveELUsingSecurityState

```
// HaveELUsingSecurityState()
// =====
// Returns TRUE if Exception level 'el' with Security state 'secure' is supported,
// FALSE otherwise.

boolean HaveELUsingSecurityState(bits(2) el, boolean secure)

    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);
        otherwise
            return (HaveEL(EL3) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

### Library pseudocode for shared/functions/system/HaveSecureState

```
// HaveSecureState()
// =====
// Return TRUE if Secure State is supported.

boolean HaveSecureState()
    if !HaveEL(EL3) then
        return SecureOnlyImplementation();
    if IsFeatureImplemented(FEAT_RME) && !IsFeatureImplemented(FEAT_SEL2) then
        return FALSE;
    return TRUE;
```

### Library pseudocode for shared/functions/system/HighestEL

```
// HighestEL()
// =====
// Returns the highest implemented Exception level.

bits(2) HighestEL()
    if HaveEL(EL3) then
        return EL3;
    elsif HaveEL(EL2) then
        return EL2;
    else
        return EL1;
```

### Library pseudocode for shared/functions/system/Hint\_CLRBHB

```
// Hint_CLRBHB()
// =====
// Provides a hint to clear the branch history for the current context.

Hint_CLRBHB();
```

### Library pseudocode for shared/functions/system/Hint\_DGH

```
// Hint_DGH()
// =====
// Provides a hint to close any gathering occurring within the implementation.

Hint_DGH();
```

### 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.

Hint_StoreShared(boolean stream);
```

## 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.

Hint_WFE()
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWfxTrap(WfxType\_WFE);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            boolean delay_enabled;
            integer delay;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired so trap WFE
                if target_el == EL3 && EL3SDDUndef() then
                    UNDEFINED;
                else
                    AArch64.WfxTrap(WfxType\_WFE, target_el);
            else
                WaitForEvent();
        else
            WaitForEvent();
```

## 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.

Hint_WFET(integer localtimeout)
    if IsEventRegisterSet() then
        ClearEventRegister();
        return;

    if IsFeatureImplemented(FEAT_WFxT) && LocalTimeoutEvent(localtimeout) then
        // No further operation if the local timeout has expired.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWfxTrap(WfxType_WFET);
    if trap then
        if IsFeatureImplemented(FEAT_TWED) then
            // Determine if trap delay is enabled and delay amount
            boolean delay_enabled;
            integer delay;
            (delay_enabled, delay) = WFETrapDelay(target_el);
            if !WaitForEventUntilDelay(delay_enabled, delay) then
                // Event did not arrive before delay expired so trap WFE
                if target_el == EL3 && EL3SDDUndef() then
                    UNDEFINED;
                else
                    AArch64.WFxTrap(WfxType_WFET, target_el);
            else
                WaitForEvent(localtimeout);
        else
            WaitForEvent(localtimeout);
```

## 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.

Hint_WFI()
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure_ERR, FALSE);

    if InterruptPending() then
        // No further operation if an interrupt is pending.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWfxTrap(WfxType_WFI);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        AArch64.WFxTrap(WfxType_WFI, target_el);
    else
        WaitForInterrupt();
```

## 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.

Hint_WFIT(integer localtimeout)
    if IsFeatureImplemented(FEAT_TME) && TSTATE.depth > 0 then
        FailTransaction(TMFailure\_ERR, FALSE);

    if (InterruptPending() || (IsFeatureImplemented(FEAT_WFxT) &&
        LocalTimeoutEvent(localtimeout))) then
        // No further operation if an interrupt is pending or the local timeout has expired.
        EndOfInstruction();
        return;

    boolean trap;
    bits(2) target_el;
    (trap, target_el) = AArch64.CheckForWFXTrap(WFXType\_WFIT);
    if trap then
        if target_el == EL3 && EL3SDDUndef() then
            UNDEFINED;
        AArch64.WFXTrap(WFXType\_WFIT, target_el);
    else
        WaitForInterrupt(localtimeout);
```

## 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.

Hint_Yield();
```

## 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.

(boolean, boolean) IRQPending();
```

## Library pseudocode for shared/functions/system/IllegalExceptionReturn

```
// IllegalExceptionReturn()
// =====

boolean IllegalExceptionReturn(bits(N) spsr)

    // 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.
    (valid, target) = ELFromSPSR(spsr);
    if !valid then return TRUE;

    // Check for return to higher Exception level
    if UInt(target) > UInt(PSTATE.EL) then return TRUE;

    spsr_mode_is_aarch32 = (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)
    (known, target_el_is_aarch32) = ELUsingAArch32K(target);
    assert known || (target == EL0 && !ELUsingAArch32(EL1));
    if known && spsr_mode_is_aarch32 != target_el_is_aarch32 then return TRUE;

    // Check for illegal return from AArch32 to AArch64
    if UsingAArch32() && !spsr_mode_is_aarch32 then return TRUE;

    // Check for illegal return to EL1 when HCR.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;

    if (IsFeatureImplemented(FEAT_GCS) && PSTATE.EXLOCK == '0' &&
        PSTATE.EL == target && GetCurrentEXLOCKEN()) then
        return TRUE;

    return FALSE;
```

## Library pseudocode for shared/functions/system/InstrSet

```
// InstrSet
// =====

enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

## Library pseudocode for shared/functions/system/InstructionFetchBarrier

```
// InstructionFetchBarrier()
// =====

InstructionFetchBarrier();
```

## Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
// InstructionSynchronizationBarrier()
// =====

InstructionSynchronizationBarrier();
```

## Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical, virtual, or delegated
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending\(\);
    (fiq_pending, -) = FIQPending\(\);
    constant boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
IsPhysicalErrorPending\(\));

    if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        constant boolean virq_pending = (HCR_EL2.IMO == '1' && (VirtualIRQPending\(\) ||
            HCR_EL2.VI == '1'));
        constant boolean vfiq_pending = (HCR_EL2.FMO == '1' && (VirtualFIQPending\(\) ||
            HCR_EL2.VF == '1'));
        constant boolean vsei_pending = ((HCR_EL2.AMO == '1' ||
            (IsFeatureImplemented(FEAT_DoubleFault2) &&
IsHCRXEL2Enabled\(\) && !ELUsingAArch32\(EL2\) &&
            HCRX_EL2.TMEA == '1')) &&
            (IsVirtualErrorPending\(\) || HCR_EL2.VSE == '1'));

        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    constant boolean pending_delegated_interrupt = (IsFeatureImplemented(FEAT_E3DSE) &&
        PSTATE.EL != EL3 && !ELUsingAArch32\(EL3\) &&
        SCR_EL3.<EndDSE,DSE> == '11');

    return pending_physical_interrupt || pending_virtual_interrupt || pending_delegated_interrupt;
```

## Library pseudocode for shared/functions/system/IsASEInstruction

```
// IsASEInstruction()
// =====
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.

boolean IsASEInstruction();
```

## Library pseudocode for shared/functions/system/IsCurrentSecurityState

```
// IsCurrentSecurityState()
// =====
// Returns TRUE if the current Security state matches
// the given Security state, and FALSE otherwise.

boolean IsCurrentSecurityState(SecurityState ss)
    return CurrentSecurityState\(\) == ss;
```

## 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.

boolean IsEventRegisterSet()
    return EventRegister == '1';
```



### Library pseudocode for shared/functions/system/IsHighestEL

```
// IsHighestEL()
// =====
// Returns TRUE if given exception level is the highest exception level implemented

boolean IsHighestEL(bits(2) el)
    return HighestEL() == el;
```

### Library pseudocode for shared/functions/system/IsInHost

```
// IsInHost()
// =====

boolean IsInHost()
    return ELIsInHost(PSTATE.EL);
```

### 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.

boolean IsSecureBelowEL3()
    if HaveEL(EL3) then
        return SCR_curr[].NS == '0';
    elseif 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 processor is Secure or FALSE if Non-secure.
        return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

### Library pseudocode for shared/functions/system/IsSecureEL2Enabled

```
// IsSecureEL2Enabled()
// =====
// Returns TRUE if Secure EL2 is enabled, FALSE otherwise.

boolean IsSecureEL2Enabled()
    if HaveEL(EL2) && IsFeatureImplemented(FEAT_SEL2) then
        if HaveEL(EL3) then
            if !ELUsingAArch32(EL3) && SCR_EL3.EEL2 == '1' then
                return TRUE;
            else
                return FALSE;
        else
            return SecureOnlyImplementation();
    else
        return FALSE;
```

## Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// LocalTimeoutEvent()
// =====
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.

boolean LocalTimeoutEvent(integer localtimeout)
    assert localtimeout >= 0;

    constant bits(64) cntvct = VirtualCounterTimer();
    if UInt(cntvct) >= localtimeout then
        return TRUE;

    IsLocalTimeoutEventPending = TRUE;
    LocalTimeoutVal = localtimeout<63:0>; // Store value to compare against
                                           // Virtual Counter Timer at subsequent clock ticks

    return FALSE;
```

## Library pseudocode for shared/functions/system/Mode

```
// Mode bits
// =====
// AArch32 PSTATE.M mode bits.

constant bits(5) M32_User      = '10000';
constant bits(5) M32_FIQ      = '10001';
constant bits(5) M32_IRQ      = '10010';
constant bits(5) M32_Svc      = '10011';
constant bits(5) M32_Monitor  = '10110';
constant bits(5) M32_Abort    = '10111';
constant bits(5) M32_Hyp      = '11010';
constant bits(5) M32_Undef    = '11011';
constant bits(5) M32_System   = '11111';
```

## Library pseudocode for shared/functions/system/NonSecureOnlyImplementation

```
// NonSecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Non-secure for this implementation.

boolean NonSecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Non-secure only implementation";
```

## Library pseudocode for shared/functions/system/PLOfEL

```
// PLOfEL()
// =====

PrivilegeLevel PLOfEL(bits(2) el)
    case el of
        when EL3    return if !HaveAArch64() then PL1 else PL3;
        when EL2    return PL2;
        when EL1    return PL1;
        when EL0    return PL0;
```

## Library pseudocode for shared/functions/system/PSTATE

```
// PSTATE
// =====
// Global per-processor state

ProcState PSTATE;
```

## Library pseudocode for shared/functions/system/PhysicalCountInt

```
// PhysicalCountInt()
// =====
// Returns the integral part of physical count value of the System counter.

bits(64) PhysicalCountInt()
    return PhysicalCount<87:24>;
```

## Library pseudocode for shared/functions/system/PrivilegeLevel

```
// PrivilegeLevel
// =====
// Privilege Level abstraction.

enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

## Library pseudocode for shared/functions/system/ProcState

```
// ProcState
// =====
// Processor state bits.
// There is no significance to the field order.

type ProcState is (
    bits (1) N,           // Negative condition flag
    bits (1) Z,           // Zero condition flag
    bits (1) C,           // Carry condition flag
    bits (1) V,           // Overflow condition flag
    bits (1) D,           // Debug mask bit [AArch64 only]
    bits (1) A,           // SError interrupt mask bit
    bits (1) I,           // IRQ mask bit
    bits (1) F,           // FIQ mask bit
    bits (1) EXLOCK,      // Lock exception return state
    bits (1) PAN,          // Privileged Access Never Bit [v8.1]
    bits (1) UAO,          // User Access Override [v8.2]
    bits (1) DIT,          // Data Independent Timing [v8.4]
    bits (1) TCO,          // Tag Check Override [v8.5, AArch64 only]
    bits (1) PM,           // PMU exception Mask
    bits (1) PPEND,        // synchronous PMU exception to be observed
    bits (2) BTYPE,        // Branch Type [v8.5]
    bits (1) PACM,         // PAC instruction modifier
    bits (1) ZA,           // Accumulation array enabled [SME]
    bits (1) SM,           // Streaming SVE mode enabled [SME]
    bits (1) ALLINT,       // Interrupt mask bit
    bits (1) UINJ,         // Undefined Exception Injection
    bits (1) SS,           // Software step bit
    bits (1) IL,           // Illegal Execution state bit
    bits (2) EL,           // Exception level
    bits (1) nRW,          // Execution state: 0=AArch64, 1=AArch32
    bits (1) SP,           // Stack pointer select: 0=SP0, 1=SPx [AArch64 only]
    bits (1) Q,           // Cumulative saturation flag [AArch32 only]
    bits (4) GE,           // Greater than or Equal flags [AArch32 only]
    bits (1) SSBS,         // Speculative Store Bypass Safe
    bits (8) IT,           // If-then bits, RES0 in CPSR [AArch32 only]
    bits (1) J,           // J bit, RES0 [AArch32 only, RES0 in SPSR and CPSR]
    bits (1) T,           // T32 bit, RES0 in CPSR [AArch32 only]
    bits (1) E,           // Endianness bit [AArch32 only]
    bits (5) M             // Mode field [AArch32 only]
)
```

## Library pseudocode for shared/functions/system/RestoredITBits

```
// RestoredITBits()
// =====
// Get the value of PSTATE.IT to be restored on this exception return.

bits(8) RestoredITBits(bits(N) spsr)
    it = 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;

    // 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';

    // 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.
    itd = if PSTATE.EL == EL2 then HSCTLR.ITD else SCTLRL.ITD;
    if (spsr<5> == '0' && !IsZero(it)) || (itd == '1' && !IsZero(it<2:0>)) then
        return '00000000';
    else
        return it;
```

## Library pseudocode for shared/functions/system/SCR\_curr

```
// SCR_curr[]
// =====

SCRType SCR_curr[]
    // AArch32 secure & AArch64 EL3 registers are not architecturally mapped
    assert HaveEL(EL3);
    bits(64) r;
    if !HaveAArch64() then
        r = ZeroExtend(SCR, 64);
    else
        r = SCR_EL3;
    return r;
```

## Library pseudocode for shared/functions/system/SecureOnlyImplementation

```
// SecureOnlyImplementation()
// =====
// Returns TRUE if the security state is always Secure for this implementation.

boolean SecureOnlyImplementation()
    return boolean IMPLEMENTATION_DEFINED "Secure-only implementation";
```

## Library pseudocode for shared/functions/system/SecurityState

```
// SecurityState
// =====
// The Security state of an execution context

enumeration SecurityState {
    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.

SecurityState SecurityStateAtEL(bits(2) EL)
    if IsFeatureImplemented(FEAT_RME) then
        if EL == EL3 then return SS\_Root;
        effective_nse_ns = SCR_EL3.NSE : EffectiveSCR\_EL3\_NS();
        case effective_nse_ns of
            when '00'
                if IsFeatureImplemented(FEAT_SEL2) then
                    return SS\_Secure;
                else
                    Unreachable();
            when '01'
                return SS\_NonSecure;
            when '11'
                return SS\_Realm;
            otherwise
                Unreachable();

    if !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS\_Secure;
        else
            return SS\_NonSecure;
    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 SCR_EL3.NS == '1' then SS\_NonSecure else SS\_Secure;
        else
            return if SCR.NS == '1' then SS\_NonSecure else SS\_Secure;
```

## 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.

SendEvent();
```

## 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.

SendEventLocal()
    EventRegister = '1';
    return;
```

## Library pseudocode for shared/functions/system/SetAccumulatedFPExceptions

```
// SetAccumulatedFPExceptions()
// =====
// Stores FP Exceptions accumulated by the PE.

SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```



```

// SetPSTATEFromPSR()
// =====

SetPSTATEFromPSR(bits(N) spsr)
    constant boolean illegal_psr_state = IllegalExceptionReturn(spsr);
    SetPSTATEFromPSR(spsr, illegal_psr_state);

// SetPSTATEFromPSR()
// =====
// Set PSTATE based on a PSR value

SetPSTATEFromPSR(bits(N) spsr_in, boolean illegal_psr_state)
    bits(N) spsr = spsr_in;
    constant boolean from_aarch64 = !UsingAArch32();
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    if IsFeatureImplemented(FEAT_SEBEP) then
        assert N == 64;
        ExceptionReturnPPEND(ZeroExtend(spsr, 64));

    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = bits(2) UNKNOWN;
        if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_PAuth_LR) then PSTATE.PACM = bit UNKNOWN;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = '0';
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if IsFeatureImplemented(FEAT_UINJ) then PSTATE.UINJ = spsr<36>;
        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>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if IsFeatureImplemented(FEAT_BTI) then PSTATE.BTYPE = spsr<11:10>;
            if IsFeatureImplemented(FEAT_SSBS) then PSTATE.SSBS = spsr<12>;
            if IsFeatureImplemented(FEAT_UAO) then PSTATE.UAO = spsr<23>;
            if IsFeatureImplemented(FEAT_DIT) then PSTATE.DIT = spsr<24>;
            if IsFeatureImplemented(FEAT_MTE) then PSTATE.TCO = spsr<25>;
            if IsFeatureImplemented(FEAT_GCS) then PSTATE.EXLOCK = spsr<34>;
            if IsFeatureImplemented(FEAT_PAuth_LR) then
                PSTATE.PACM = if IsPACMEnabled() then spsr<35> else '0';

// 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';

// 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>;
if PSTATE.nRW == '1' then // AArch32 state
    PSTATE.Q = spsr<27>;
    PSTATE.IT = RestoredITBits(spsr);
    ShouldAdvanceIT = FALSE;
    if IsFeatureImplemented(FEAT_DIT) then
        PSTATE.DIT = (if (Restarting() || from_aarch64) then spsr<24> else spsr<21>);
    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>;
        if IsFeatureImplemented(FEAT_NMI) then PSTATE.ALLINT = spsr<13>;
        PSTATE.<D,A,I,F> = spsr<9:6>;           // No PSTATE.<Q,IT,GE,E,T> in AArch64 state
    return;

```

### Library pseudocode for shared/functions/system/ShouldAdvanceHS

```

// ShouldAdvanceHS
// =====
// Cleared if we should not advance the EDESR.SS after the current instruction.

boolean ShouldAdvanceHS;

```

### Library pseudocode for shared/functions/system/ShouldAdvanceIT

```

// ShouldAdvanceIT
// =====
// Cleared if we should not advance the PSTATE.IT after the current instruction.

boolean ShouldAdvanceIT;

```

### Library pseudocode for shared/functions/system/ShouldAdvanceSS

```

// ShouldAdvanceSS
// =====
// Cleared if PSTATE.SS is written by the current instruction.

boolean ShouldAdvanceSS;

```

### Library pseudocode for shared/functions/system/ShouldSetPPEND

```

// ShouldSetPPEND
// =====
// TRUE if PSTATE.PPEND is set or cleared at the end of the current instruction, according to
// whether a PMU counter configured for synchronous mode overflowed or not.
// Otherwise, PSTATE.PPEND is not changed at the end of the instruction.

boolean ShouldSetPPEND;

```

### Library pseudocode for shared/functions/system/SmallestTranslationGranule

```

// SmallestTranslationGranule()
// =====
// Smallest implemented translation granule.

integer SmallestTranslationGranule()
    if IsFeatureImplemented(FEAT_TGran4K) then return 12;
    if IsFeatureImplemented(FEAT_TGran16K) then return 14;
    if IsFeatureImplemented(FEAT_TGran64K) then return 16;
    Unreachable();

```

### Library pseudocode for shared/functions/system/SpeculationBarrier

```

// SpeculationBarrier()
// =====

SpeculationBarrier();

```



### Library pseudocode for shared/functions/system/SyncCounterOverflowed

```
// SyncCounterOverflowed
// =====
// Set if a PMU counter configured for synchronous mode has overflowed
// during the execution of the current instruction.

boolean SyncCounterOverflowed;
```

### Library pseudocode for shared/functions/system/SynchronizeContext

```
// SynchronizeContext()
// =====
// Context Synchronization event, includes Instruction Fetch Barrier effect

SynchronizeContext();
```

### Library pseudocode for shared/functions/system/SynchronizeErrors

```
// SynchronizeErrors()
// =====
// Implements the error synchronization event.

SynchronizeErrors();
```

### Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// TakeUnmaskedPhysicalSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt.

TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

### Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// TakeUnmaskedSErrorInterrupts()
// =====
// Take any pending unmasked physical SError interrupt or unmasked virtual SError
// interrupt.

TakeUnmaskedSErrorInterrupts();
```

### Library pseudocode for shared/functions/system/ThisInstr

```
// ThisInstr()
// =====

bits(32) ThisInstr();
```

### Library pseudocode for shared/functions/system/ThisInstrLength

```
// ThisInstrLength()
// =====

integer ThisInstrLength();
```

### Library pseudocode for shared/functions/system/UndefinedInjectionCheck

```
// UndefinedInjectionCheck()
// =====
// Check PSTATE.UINJ to determine if execution of the current
// instruction should cause an Undefined exception.

UndefinedInjectionCheck()
    if IsFeatureImplemented(FEAT_UINJ) && PSTATE.UINJ == '1' then
        UNDEFINED;
```

### Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()
// =====
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.

boolean UsingAArch32()
    constant boolean aarch32 = (PSTATE.nRW == '1');
    if !HaveAArch32() then assert !aarch32;
    if !HaveAArch64() then assert aarch32;
    return aarch32;
```

### 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.

boolean ValidSecurityStateAtEL(bits(2) el)
    if !HaveEL(el) then
        return FALSE;

    if el == EL3 then
        return TRUE;

    if IsFeatureImplemented(FEAT_RME) then
        constant bits(2) effective_nse_ns = SCR_EL3.NSE : EffectiveSCR_EL3_NS();
        if effective_nse_ns == '10' then
            return FALSE;

    if el == EL2 then
        return EL2Enabled();

    return TRUE;
```

### Library pseudocode for shared/functions/system/VirtualFIQPending

```
// VirtualFIQPending()
// =====
// Returns TRUE if there is any pending virtual FIQ.

boolean VirtualFIQPending();
```

### Library pseudocode for shared/functions/system/VirtualIRQPending

```
// VirtualIRQPending()
// =====
// Returns TRUE if there is any pending virtual IRQ.

boolean VirtualIRQPending();
```

## Library pseudocode for shared/functions/system/WFxType

```
// WFxType
// =====
// WFx instruction types.

enumeration WFxType {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.

WaitForEvent()
    if Halted() then return;
    if !IsEventRegisterSet() then
        EnterLowPowerState();
    return;

// 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.

WaitForEvent(integer localtimeout)
    if Halted() then return;
    if !(IsEventRegisterSet() || LocalTimeoutEvent(localtimeout)) then
        EnterLowPowerState();
    return;
```

## 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.

WaitForInterrupt()
    if Halted() then return;
    EnterLowPowerState();
    return;

// 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.

WaitForInterrupt(integer localtimeout)
    if Halted() then return;
    if !LocalTimeoutEvent(localtimeout) then
        EnterLowPowerState();
    return;
```

## Library pseudocode for shared/functions/system/WatchpointRelatedSyndrome

```
// WatchpointRelatedSyndrome()
// =====
// Update common Watchpoint related fields.

bits(24) WatchpointRelatedSyndrome(FaultRecord fault)
    bits(24) syndrome = Zeros(24);

    if fault.watchptinfo.maybe_false_match then
        syndrome<16> = '1'; // WPF
    elsif IsFeatureImplemented(FEAT_Debugv8p2) then
        syndrome<16> = bit IMPLEMENTATION_DEFINED "WPF value on TRUE Watchpoint match";

    if IsRelaxedWatchpointAccess(fault.accessdesc) then
        if HaltOnBreakpointOrWatchpoint() then
            if boolean IMPLEMENTATION_DEFINED "EDWAR is not valid on watchpoint debug event" then
                syndrome<10> = '1'; // FnV
            else
                if boolean IMPLEMENTATION_DEFINED "FAR is not valid on watchpoint exception" then
                    syndrome<10> = '1'; // FnV
        else
            if WatchpointFARNotPrecise(fault) then
                syndrome<15> = '1'; // FnP

    // 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 AddressNotInNaturallyAlignedBlock(fault.vaddress) then // WPTV
            syndrome<17> = '1';
        elsif boolean IMPLEMENTATION_DEFINED "WPTV field is valid" then
            syndrome<17> = '1';
        if syndrome<17> == '1' then
            syndrome<23:18> = fault.watchptinfo.watchpt_num<5:0>; // WPT
        else
            syndrome<23:18> = bits(6) UNKNOWN;

    return syndrome;
```

## Library pseudocode for shared/functions/unbounded/Unbounded

```
// Unbounded
// =====
// Returns an upper limit for a specific loop or function.

integer Unbounded(string s);
```

## Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictable

```
// 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.)

Constraint ConstrainUnpredictable(Unpredictable which);
```

## 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.

(Constraint,bits(width)) ConstrainUnpredictableBits(Unpredictable which, integer width);
```

## 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.

boolean ConstrainUnpredictableBool(Unpredictable which);
```

## 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.

(Constraint,integer) ConstrainUnpredictableInteger(integer low, integer high,
Unpredictable which);
```

## 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.

ConstrainUnpredictableProcedure(Unpredictable which);
```

## Library pseudocode for shared/functions/unpredictable/Constraint

```
// Constraint
// =====
// List of Constrained Unpredictable behaviors.

enumeration Constraint    { // 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_UNDEFEL0,  // 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

    // IPA too large
    Constraint_FORCE, Constraint_FORCENOSLDCHECK,
    // An unallocated System register value maps onto an allocated value
    Constraint_MAPTOALLOCATED,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT
};
```



```

// Unpredictable
// =====
// List of Constrained Unpredictable situations.

enumeration Unpredictable {
    // 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_RESPRRR,
    // 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_16BYTEROUNDEDOWNACCESS,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Dirty Bit State Update by HW
Unpredictable_DBUPDATE,
// Consider SCTL_R_ELx[].IESB in Debug state
Unpredictable_IESBinDebug,
// Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
Unpredictable_BADPMSFCR,

```

```

// Zero saved BType value in SPSR_ELx/DPSR_EL0
Unpredictable_ZEROBTYPE,
// Timestamp constrained to virtual or physical
Unpredictable_EL2TIMESTAMP,
Unpredictable_EL1TIMESTAMP,
// Reserved MDCR_EL3.<NSTBE,NSTB> or MDCR_EL3.<NSPBE,NSPB> value
Unpredictable_RESERVEDNSxB,
// 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_CLEARERRITZZERO,
// ALUEXCEPTIONRETURN when in user/system mode in
// A32 instructions
Unpredictable_ALUEXCEPTIONRETURN,
// Trap to register in debug state are ignored
Unpredictable_IGNORETRAPINDEBUG,
// Compare DBGxVR.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,
// 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_RESEPMSAD,
// 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_WRITEFAILEDCCAS,
// Store-only Tag checking on a failed Atomic Compare and Swap
Unpredictable_STOREONLYTAGCHECKEDCCAS,
// 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,
// 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_MMU_IMPDEF_FAULT,
Unpredictable_Atomic_NOP,
// Accessing DBGDSRint via MRC in debug state
Unpredictable_MRC_APSR_TARGET
};

```

## Library pseudocode for shared/functions/vector/AdvSIMDExpandImm

```

// AdvSIMDExpandImm()
// =====

bits(64) AdvSIMDExpandImm(bit op, bits(4) cmode, bits(8) imm8)
bits(64) imm64;
case cmode<3:1> of
when '000'
    imm64 = Replicate(Zeros(24):imm8, 2);
when '001'
    imm64 = Replicate(Zeros(16):imm8:Zeros(8), 2);
when '010'
    imm64 = Replicate(Zeros(8):imm8:Zeros(16), 2);
when '011'
    imm64 = Replicate(imm8:Zeros(24), 2);
when '100'
    imm64 = Replicate(Zeros(8):imm8, 4);
when '101'
    imm64 = Replicate(imm8:Zeros(8), 4);
when '110'
    if cmode<0> == '0' then
        imm64 = Replicate(Zeros(16):imm8:Ones(8), 2);
    else
        imm64 = Replicate(Zeros(8):imm8:Ones(16), 2);
when '111'
    if cmode<0> == '0' && op == '0' then
        imm64 = Replicate(imm8, 8);
    if cmode<0> == '0' && op == '1' then
        imm8a = Replicate(imm8<7>, 8); imm8b = Replicate(imm8<6>, 8);
        imm8c = Replicate(imm8<5>, 8); imm8d = Replicate(imm8<4>, 8);
        imm8e = Replicate(imm8<3>, 8); imm8f = Replicate(imm8<2>, 8);
        imm8g = Replicate(imm8<1>, 8); imm8h = Replicate(imm8<0>, 8);
        imm64 = imm8a:imm8b:imm8c:imm8d:imm8e:imm8f:imm8g:imm8h;
    if cmode<0> == '1' && op == '0' then
        imm32 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 5):imm8<5:0>:Zeros(19);
        imm64 = Replicate(imm32, 2);
    if cmode<0> == '1' && op == '1' then
        if UsingAArch32() then ReservedEncoding();
        imm64 = imm8<7>:NOT(imm8<6>):Replicate(imm8<6>, 8):imm8<5:0>:Zeros(48);

return imm64;

```

## 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])

bits(N) MatMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, boolean op1_unsigned,
                  boolean op2_unsigned)
    assert N == 128;

    bits(N) result;
    bits(32) sum;
    integer prod;

    for i = 0 to 1
        for j = 0 to 1
            sum = Elem[addend, 2*i + j, 32];
            for k = 0 to 7
                prod = (Int(Elem[op1, 8*i + k, 8], op1_unsigned) *
                       Int(Elem[op2, 8*j + k, 8], op2_unsigned));
                sum = sum + prod;
            Elem[result, 2*i + j, 32] = sum;

    return result;
```

## Library pseudocode for shared/functions/vector/PolynomialMult

```
// PolynomialMult()
// =====

bits(M+N) PolynomialMult(bits(M) op1, bits(N) op2)
    result = Zeros(M+N);
    extended_op2 = ZeroExtend(op2, M+N);
    for i=0 to M-1
        if op1[i] == '1' then
            result = result EOR LSL(extended_op2, i);
    return result;
```

## Library pseudocode for shared/functions/vector/SatQ

```
// SatQ()
// =====

(bits(N), boolean) SatQ(integer i, integer N, boolean unsigned)
    (result, sat) = if unsigned then UnsignedSatQ(i, N) else SignedSatQ(i, N);
    return (result, sat);
```

## Library pseudocode for shared/functions/vector/SignedSatQ

```
// SignedSatQ()
// =====

(bits(N), boolean) SignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    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;
    return (result<N-1:0>, saturated);
```

## Library pseudocode for shared/functions/vector/UnsignedRSqrtEstimate

```
// UnsignedRSqrtEstimate()
// =====

bits(N) UnsignedRSqrtEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    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)
        increasedprecision = FALSE;
        estimate = 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);

    return result;
```

## Library pseudocode for shared/functions/vector/UnsignedRecipEstimate

```
// UnsignedRecipEstimate()
// =====

bits(N) UnsignedRecipEstimate(bits(N) operand)
    assert N == 32;
    bits(N) result;
    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)
        increasedprecision = FALSE;
        estimate = 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);

    return result;
```

## Library pseudocode for shared/functions/vector/UnsignedSatQ

```
// UnsignedSatQ()
// =====

(bits(N), boolean) UnsignedSatQ(integer i, integer N)
    integer result;
    boolean saturated;
    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;
    return (result<N-1:0>, saturated);
```

## 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.

(PhysMemRetStatus, AddressDescriptor) DebugMemWrite(bits(64) vaddress, AccessDescriptor accdesc,
                                                    boolean aligned, bits(8) data)

    PhysMemRetStatus memstatus = PhysMemRetStatus UNKNOWN;

    // Translate virtual address
    AddressDescriptor addrdesc;
    constant integer size = 1;
    addrdesc = AArch64.TranslateAddress(vaddress, accdesc, aligned, size);

    if IsFault(addrdesc) then
        return (memstatus, addrdesc);

    memstatus = PhysMemWrite(addrdesc, 1, accdesc, data);

    return (memstatus, addrdesc);
```



```

// DebugWriteExternalAbort()
// =====
// Populate the syndrome register for an External abort caused by a call of DebugMemWrite().

DebugWriteExternalAbort(PhysMemRetStatus memstatus, AddressDescriptor addrdesc,
                        bits(64) start_vaddr)

    constant boolean iswrite = TRUE;

    boolean handle_as_SError = FALSE;
    boolean async_external_abort = FALSE;
    bits(64) syndrome;
    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "SPE SyncExternal as SError";
            async_external_abort = boolean IMPLEMENTATION_DEFINED "SPE async External abort";
            syndrome = PMBSR_EL1<63:0>;
        when AccessType\_TRBE
            handle_as_SError = boolean IMPLEMENTATION_DEFINED "TRBE SyncExternal as SError";
            async_external_abort = boolean IMPLEMENTATION_DEFINED "TRBE async External abort";
            syndrome = TRBSR_EL1<63:0>;
        otherwise
            Unreachable();

    boolean ttw_abort;
    ttw_abort = addrdesc.fault.statuscode IN {Fault\_SyncExternalOnWalk,
                                              Fault\_SyncParityOnWalk};
    constant Fault statuscode = (if ttw_abort then addrdesc.fault.statuscode
                                else memstatus.statuscode);
    constant bit extflag = if ttw_abort then addrdesc.fault.extflag else memstatus.extflag;
    if (statuscode IN {Fault\_AsyncExternal, Fault\_AsyncParity} || handle_as_SError) then
        // ASYNC Fault -> SError or SYNC Fault handled as SError
        FaultRecord fault = NoFault();
        constant boolean parity = 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;
        fault.extflag = extflag;
        fault.accessdesc.acctype = addrdesc.fault.accessdesc.acctype;
        PendSErrorInterrupt(fault);
    else
        // SYNC Fault, not handled by SError
        // Generate Buffer Management Event
        // EA bit
        syndrome<18> = '1';

        // DL bit for SPE
        if addrdesc.fault.accessdesc.acctype == AccessType\_SPE && (async_external_abort ||
            (start_vaddr != addrdesc.vaddress)) then
            syndrome<19> = '1';

        // Do not change following values if previous Buffer Management Event
        // has not been handled.
        // S bit
        if IsZero(syndrome<17>) then
            syndrome<17> = '1';

        // EC bits
        bits(6) ec;
        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';
        syndrome<31:26> = ec;

        // MSS bits
        if async_external_abort then
            syndrome<15:0> = Zeros(10) : '010001';

```



```

        else
            syndrome<15:0> = Zeros(10) : EncodeLDFSC(statuscode, addrdesc.fault.level);

    case addrdesc.fault.accessdesc.acctype of
        when AccessType\_SPE
            PMBSR_EL1<63:0> = syndrome;
        when AccessType\_TRBE
            //IRQ
            syndrome<22> = '1';

            TRBSR_EL1<63:0> = syndrome;
        otherwise
            Unreachable();

```

## Library pseudocode for shared/trace/Common/DebugWriteFault

```

// DebugWriteFault()
// =====
// Populate the syndrome register for a fault caused by a call of DebugMemWrite().

DebugWriteFault(bits(64) vaddress, FaultRecord fault)
    bits(16) mss = Zeros(16);

    // FSC
    mss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    // EC bits
    bits(6) ec;
    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';

    bits(24) mss2 = Zeros(24);
    if fault.statuscode == Fault\_Permission then
        // AssuredOnly
        mss2<7> = if fault.assuredonly then '1' else '0';
        // Overlay
        mss2<6> = if fault.overlay then '1' else '0';
        // DirtyBit
        mss2<5> = if fault.dirtybit then '1' else '0';

    bits(2) target_el;
    case fault.accessdesc.acctype of
        when AccessType\_SPE
            target_el = ReportSPEEvent(ec, mss<5:0>);
            PMBSR\_EL[target_el].S = '1';
            PMBSR\_EL[target_el].EC = ec;
            PMBSR\_EL[target_el].MSS = mss;
            PMBSR\_EL[target_el].MSS2 = mss2;
            // Buffer Write Pointer already points to the address that generated the fault.
            // Writing to memory never started so no data loss. DL is unchanged.

        when AccessType\_TRBE
            target_el = ReportTRBEEEvent(ec, mss<5:0>);
            TRBSR\_EL[target_el].S = '1';
            TRBSR\_EL[target_el].IRQ = '1';
            TRBSR\_EL[target_el].EC = ec;
            TRBSR\_EL[target_el].MSS = mss;
            TRBSR\_EL[target_el].MSS2 = mss2;

        otherwise
            Unreachable();

    return;

```

## Library pseudocode for shared/trace/Common/GetTimestamp

```
// GetTimestamp()
// =====
// Returns the Timestamp depending on the type

bits(64) GetTimestamp(TimeStamp timeStampType)
    case timeStampType of
        when TimeStamp\_Physical
            return PhysicalCountInt();
        when TimeStamp\_Virtual
            return PhysicalCountInt() - CNTVOFF_EL2;
        when TimeStamp\_OffsetPhysical
            constant bits(64) physoff = if PhysicalOffsetIsValid() then CNTPOFF_EL2 else Zeros(64);
            return PhysicalCountInt() - physoff;
        when TimeStamp\_None
            return Zeros(64);
        when TimeStamp\_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

## Library pseudocode for shared/trace/Common/PhysicalOffsetIsValid

```
// PhysicalOffsetIsValid()
// =====
// Returns whether the Physical offset for the timestamp is valid

boolean PhysicalOffsetIsValid()
    if !HaveAArch64() then
        return FALSE;
    elsif !HaveEL(EL2) || !IsFeatureImplemented(FEAT_ECV_POFF) then
        return FALSE;
    elsif HaveEL(EL3) && SCR_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;
```

## Library pseudocode for shared/trace/TraceBranch/BranchNotTaken

```
// BranchNotTaken()
// =====
// Called when a branch is not taken.

BranchNotTaken(BranchType branchtype, boolean branch_conditional)
    constant boolean branchtaken = FALSE;
    if IsFeatureImplemented(FEAT_SPE) then
        SPEBranch(bits(64) UNKNOWN, branchtype, branch_conditional, branchtaken);
    return;
```

## Library pseudocode for shared/trace/TraceBuffer/AllowExternalTraceBufferAccess

```
// AllowExternalTraceBufferAccess()
// =====
// Returns TRUE if an external debug interface access to the Trace Buffer
// registers is allowed, FALSE otherwise.
// The access may also be subject to OS Lock, power-down, etc.

boolean AllowExternalTraceBufferAccess()
    return AllowExternalTraceBufferAccess(AccessState());

// 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.

boolean AllowExternalTraceBufferAccess(SecurityState access_state)
    assert IsFeatureImplemented(FEAT_TRBE_EXT);
    // FEAT_Debugv8p4 is always implemented when FEAT_TRBE_EXT is implemented.
    assert IsFeatureImplemented(FEAT_Debugv8p4);

    bits(2) etbad = if HaveEL(EL3) then MDCR_EL3.ETBAD else '11';

    // Check for reserved values
    if !IsFeatureImplemented(FEAT_RME) && etbad IN {'01','10'} then
        (-, etbad) = ConstrainUnpredictableBits(Unpredictable\_RES\_ETBAD, 2);
        // The value returned by ConstrainUnpredictableBits must be a
        // non-reserved value
        assert etbad IN {'00', '11'};

    case etbad of
        when '00'
            if IsFeatureImplemented(FEAT_RME) then
                return access_state == SS\_Root;
            else
                return access_state == SS\_Secure;
        when '01'
            assert IsFeatureImplemented(FEAT_RME);
            return access_state IN {SS\_Root, SS\_Realm};
        when '10'
            assert IsFeatureImplemented(FEAT_RME);
            return access_state IN {SS\_Root, SS\_Secure};
        when '11'
            return TRUE;
```

## Library pseudocode for shared/trace/TraceBuffer/CheckForTRBEEException

```
// CheckForTRBEEException()
// =====
// Take a TRBE Profiling exception if pending, permitted, and unmasked.

CheckForTRBEEException()
    if !IsFeatureImplemented(FEAT_TRBE_EXC) && !SelfHostedTraceEnabled() then
        return;

    if Halted() || Restarting() then
        return;

    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;
    boolean route_to_el1 = FALSE;

    if HaveEL(EL3) && MDCR_EL3.TRBEE == '1x' then
        constant boolean pending = TRBSR_EL3.IRQ == '1';
        constant boolean masked = PSTATE.EL == EL3;
        route_to_el3 = pending && !masked;

    SecurityState owning_ss;
    bits(2) owning_el;
    (owning_ss, owning_el) = TraceBufferOwner();
    constant boolean in_owning_ss = IsCurrentSecurityState(owning_ss);

    if EffectiveTRFCR_EL2_EE() IN {'1x'} then
        boolean pending = TRBSR_EL2.IRQ == '1';
        boolean masked = (!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;

    if EffectiveTRFCR_EL1_EE() == '11' then
        constant boolean pending = TRBSR_EL1.IRQ == '1';
        constant boolean masked = (!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;

    constant bits(5) fsc = '00010'; // TRBE exception
    constant boolean synchronous = FALSE;

    // The relative priorities of the following checks is IMPLEMENTATION DEFINED
    if route_to_el3 then
        TakeProfilingException(EL3, fsc, synchronous);
    if route_to_el2 then
        TakeProfilingException(EL2, fsc, synchronous);
    if route_to_el1 then
        TakeProfilingException(EL1, fsc, synchronous);
```



```

// 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.

boolean CollectTrace(bits(8) datum)

    if !TraceBufferEnabled() then // Trace buffer disabled
        // 'datum' is discarded
        if HaveImpDefTraceOutput() then
            return ImpDefTraceOutput(datum);
        else
            return TRUE; // Discard the trace byte

    // If the TRBE cannot accept the trace data, it must return FALSE
    if TRBEInternalBufferFull() then
        return FALSE;

    if TraceBufferRunning() then // Accept the data
        address = TRBPTR_EL1;
        AddressDescriptor addrdesc;
        PhysMemRetStatus memstatus;
        FaultRecord fault;
        (owning_ss, owning_el) = TraceBufferOwner();

        aligned = TRUE;
        constant AccessDescriptor accdesc = CreateAccDescTRBE(owning_ss, owning_el);
        (memstatus, addrdesc) = DebugMemWrite(address, accdesc, aligned, datum);

        fault = addrdesc.fault;
        boolean ttw_fault;
        boolean ttw_fault_as_external_abort;
        ttw_fault = fault.statuscode IN {Fault_SyncExternalOnWalk, Fault_SyncParityOnWalk};
        ttw_fault_as_external_abort = (boolean IMPLEMENTATION_DEFINED
            "TRBE TTW fault External abort");

        if IsFault(fault.statuscode) && !(ttw_fault && ttw_fault_as_external_abort) then
            DebugWriteFault(address, fault);
            TryAssertTRBIRQ();
            return TRUE;
        elsif IsFault(memstatus) || (ttw_fault && ttw_fault_as_external_abort) then
            DebugWriteExternalAbort(memstatus, addrdesc, address);
            TryAssertTRBIRQ();
            return TRUE;

    // Check for Trigger Event
    constant bits(2) target_el = DefaultTRBEEEvent();
    constant boolean triggered = 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();

    // Increment the pointer
    next_address = 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';
        if TRBLIMITR_EL1.FM == '00' then // Fill mode
            constant bits(6) bsc = '000001'; // Buffer full event
            OtherTRBEManagementEvent(bsc);
        elsif TRBLIMITR_EL1.FM != '11' then // Not Circular Buffer mode
            TRBSR_EL[target_el].IRQ = '1'; // Assert interrupt or exception
        TRBPTR_EL1 = next_address < 63:0>;

    TryAssertTRBIRQ();
return TRUE;

```

## 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.

bits(2) DefaultTRBEEEvent()
    return ReportTRBEEEvent(Zeros(6), bits(6) UNKNOWN);
```

## Library pseudocode for shared/trace/TraceBuffer/DetectedTraceTrigger

```
// DetectedTraceTrigger()
// =====
// Called when the trace unit detects a trace trigger

DetectedTraceTrigger()
    if TraceBufferRunning() then
        constant bits(2) target_el = DefaultTRBEEEvent();
        if TRBSR_EL[target_el].TRG == '0' then
            TRBSR\_EL[target_el].TRG = '1';
            if IsZero(TRBTGR_EL1.TRG) && TRBLIMITR_EL1.TM != '11' then
                TraceUnitFlush();
                TraceUnitFlushOnTriggerComplete();
```

## Library pseudocode for shared/trace/TraceBuffer/EffectiveTRBLIMITR\_EL1\_nVM

```
// EffectiveTRBLIMITR_EL1_nVM()
// =====

bit EffectiveTRBLIMITR_EL1_nVM()
    if IsFeatureImplemented(FEAT_TRBEv1p1) && HaveEL(EL2) then
        (owning_ss, owning_el) = TraceBufferOwner();
        if ((owning_ss != SS\_Secure || IsSecureEL2Enabled()) && owning_el == EL1 &&
            TRFCR_EL2.DnVM == '1') then
            return '0';
    return TRBLIMITR_EL1.nVM;
```

## 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.

bits(2) EffectiveTRFCR_EL1_EE()
    if EffectiveTRFCR\_EL2\_EE() == '00' then
        return '00';

    bits(2) ee = TRFCR_EL1.EE;
    if ee IN {'01', '10'} then // Reserved value
        if IsFeatureImplemented(FEAT_NV) then
            ee<0> = ee<1>;
        else
            Constraint c;
            (c, ee) = ConstrainUnpredictableBits(Unpredictable\_RESTRFEE, 2);
            assert c IN {Constraint\_DISABLED, Constraint\_UNKNOWN};
            if c == Constraint\_DISABLED then
                ee = '00';
            // Otherwise the value returned by ConstrainUnpredictableBits must be
            // a non-reserved value

    return ee;
```

## Library pseudocode for shared/trace/TraceBuffer/EffectiveTRFCR\_EL2\_EE

```
// EffectiveTRFCR_EL2_EE()
// =====
// Return the Effective value of TRFCR_EL2.EE.

bits(2) EffectiveTRFCR_EL2_EE()
    if !IsFeatureImplemented(FEAT_TRBE_EXC) || !SelfHostedTraceEnabled() then
        return '00';

    if HaveEL(EL3) && MDCR_EL3.TRBEE == '00' then
        return '00';

    constant boolean check_el2 = HaveEL(EL2) && (EffectiveSCR_EL3_NS() == '1' ||
        IsSecureEL2Enabled());
    return if check_el2 then TRFCR_EL2.EE else '01';
```

## Library pseudocode for shared/trace/TraceBuffer/GetTRBSR\_EL1\_FSC

```
// GetTRBSR_EL1_FSC()
// =====
// Query the TRBSR_EL1.FSC field.

bits(6) GetTRBSR_EL1_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL1<5:0>;
    return FSC;
```

## Library pseudocode for shared/trace/TraceBuffer/GetTRBSR\_EL2\_FSC

```
// GetTRBSR_EL2_FSC()
// =====
// Query the TRBSR_EL2.FSC field.

bits(6) GetTRBSR_EL2_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL2<5:0>;
    return FSC;
```

## Library pseudocode for shared/trace/TraceBuffer/GetTRBSR\_EL3\_FSC

```
// GetTRBSR_EL3_FSC()
// =====
// Query the TRBSR_EL3.FSC field.

bits(6) GetTRBSR_EL3_FSC()
    bits(6) FSC;

    FSC = TRBSR_EL3<5:0>;
    return FSC;
```

## Library pseudocode for shared/trace/TraceBuffer/HaveImpDefTraceOutput

```
// HaveImpDefTraceOutput()
// =====

boolean HaveImpDefTraceOutput()
    return boolean IMPLEMENTATION_DEFINED "Has Enabled External Trace Port";
```



## Library pseudocode for shared/trace/TraceBuffer/ImpDefTraceOutput

```
// ImpDefTraceOutput()
// =====

boolean ImpDefTraceOutput(bits(8) datum)
    // Send 'datum' to an IMPLEMENTATION DEFINED trace output port
    // return TRUE if the byte is sent
    return FALSE;
```

## Library pseudocode for shared/trace/TraceBuffer/OtherTRBEManagementEvent

```
// OtherTRBEManagementEvent()
// =====
// Report an Other buffer management event, with the status code 'bsc'

OtherTRBEManagementEvent(bits(6) bsc)

    constant bits(2) target_el = DefaultTRBEEvent\(\);
    if TRBSR_EL[target_el].S == '0' then
        TRBSR\_EL[target_el].S = '1';           // Stop collection
        TRBSR\_EL[target_el].IRQ = '1';         // Assert interrupt or exception
        TRBSR\_EL[target_el].EC = '000000';     // Other buffer management event
        TRBSR\_EL[target_el].MSS = ZeroExtend(bsc, 16);
        TRBSR\_EL[target_el].MSS2 = Zeros(24);
```

## Library pseudocode for shared/trace/TraceBuffer/ReportTRBEEEvent

```
// ReportTRBEEvent()
// =====
// 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().

bits(2) ReportTRBEEvent(bits(6) ec_bits, bits(6) fsc_bits)
    bits(2) target_el;
    boolean route_to_el3 = FALSE;
    boolean route_to_el2 = FALSE;

    if IsFeatureImplemented(FEAT_TRBE_EXC) && SelfHostedTraceEnabled() then
        constant boolean s1fault = (ec_bits == '100100'); // Stage 1 fault
        constant boolean s2fault = (ec_bits == '100101'); // Stage 2 fault

        boolean gpcfault, gpfault;
        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;

        SecurityState owning_ss;
        bits(2) owning_el;
        (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'));

        if EffectiveTRFCR_EL2_EE() == '1x' then
            route_to_el2 = (TRFCR_EL2.EE == '11' || (s1fault && owning_el == EL2) || s2fault ||
                            gpcfault || (gpfault && HCR_EL2.GPF == '1'));

        if route_to_el3 then
            target_el = EL3;
        elsif route_to_el2 then
            target_el = EL2;
        else
            target_el = EL1;

    return target_el;
```

## Library pseudocode for shared/trace/TraceBuffer/TRBEInternalBufferFull

```
// TRBEInternalBufferFull()
// =====

boolean TRBEInternalBufferFull()
    // In the simple sequential model, the internal buffer never fills
    return FALSE;
```

## Library pseudocode for shared/trace/TraceBuffer/TRBEInterruptEnabled

```
// TRBEInterruptEnabled()
// =====
// Return TRUE if the TRBE interrupt request (TRBIRQ) is enabled, FALSE otherwise.

boolean TRBEInterruptEnabled()
    return EffectiveTRFCR_EL1_EE() == '00';
```

## 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

bit TRBE_TRBIDR_P_Read()
    SecurityState owning_ss;
    bits(2) owning_el;
    (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';
```

## Library pseudocode for shared/trace/TraceBuffer/TRBSR\_EL

```
// TRBSR_EL[] - assignment form
// =====

TRBSRType TRBSR_EL[bits(2) el]
    bits(64) r;
    case el of
        when EL1    r = TRBSR_EL1;
        when EL2    r = TRBSR_EL2;
        when EL3    r = TRBSR_EL3;
        otherwise   Unreachable();
    return r;

// TRBSR_EL[] - non-assignment form
// =====

TRBSR_EL[bits(2) el] = bits(64) value
    constant bits(64) r = value;
    case el of
        when EL1    TRBSR_EL1 = r;
        when EL2    TRBSR_EL2 = r;
        when EL3    TRBSR_EL3 = r;
        otherwise   Unreachable();
    return;
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferEnabled

```
// TraceBufferEnabled()
// =====

boolean TraceBufferEnabled()
    if !IsFeatureImplemented(FEAT_TRBE) || TRBLIMITR_EL1.E == '0' then
        return FALSE;
    if !SelfHostedTraceEnabled() then
        return FALSE;
    (-, el) = TraceBufferOwner();
    return !ELUsingAArch32(el);
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferOwner

```
// TraceBufferOwner()
// =====
// Return the owning Security state and Exception level. Must only be called
// when SelfHostedTraceEnabled() is TRUE.

(SecurityState, bits(2)) TraceBufferOwner()
    assert IsFeatureImplemented(FEAT_TRBE) && SelfHostedTraceEnabled\(\);

    SecurityState owning_ss;
    if HaveEL\(EL3\) then
        bits(3) state_bits;
        if IsFeatureImplemented(FEAT_RME) then
            state_bits = MDCR_EL3.<NSTBE,NSTB>;
            if (state_bits == '10x' ||
                (!IsFeatureImplemented(FEAT_SEL2) && state_bits == '00x')) then
                // Reserved value
                (-, state_bits) = ConstrainUnpredictableBits\(Unpredictable\_RESERVEDNSxB, 3\);
        else
            state_bits = '0' : MDCR_EL3.NSTB;

        case state_bits of
            when '00x' owning_ss = SS\_Secure;
            when '01x' owning_ss = SS\_NonSecure;
            when '11x' owning_ss = SS\_Realm;
        else
            owning_ss = if SecureOnlyImplementation\(\) then SS\_Secure else SS\_NonSecure;
    bits(2) owning_el;
    if HaveEL\(EL2\) && (owning_ss != SS\_Secure || IsSecureEL2Enabled\(\)) then
        owning_el = if MDCR_EL2.E2TB == '00' then EL2 else EL1;
    else
        owning_el = EL1;
    return (owning_ss, owning_el);
```

## Library pseudocode for shared/trace/TraceBuffer/TraceBufferRunning

```
// TraceBufferRunning()
// =====

boolean TraceBufferRunning()
    if !TraceBufferEnabled\(\) then
        return FALSE;

    boolean stopped = 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');
        if EffectiveTRFCR\_EL2\_EE\(\) == '1x' then
            stopped = stopped || (TRBSR_EL2.S == '1');
    return !stopped;
```

## 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.

TraceUnitFlushOnTriggerComplete()
    if TRBLIMITR_EL1.TM == '00' then // Stop on trigger
        constant bits(6) bsc = '000010'; // Trigger event
        OtherTRBEManagementEvent(bsc);
    elsif TRBLIMITR_EL1.TM != '11' then // Not Ignore trigger
        constant bits(2) target_el = DefaultTRBEEEvent\(\);
        TRBSR\_EL1[target_el].IRQ = '1'; // Assert interrupt or exception
```

## Library pseudocode for shared/trace/TraceBuffer/TryAssertTRBIRQ

```
// TryAssertTRBIRQ()
// =====
// Assert TRBIRQ pin when appropriate.

TryAssertTRBIRQ()
    if TRBEInterruptEnabled() then
        SetInterruptRequestLevel(InterruptID\_TRBIRQ,
                                if TRBSR_EL1.IRQ == '1' then Signal\_High else Signal\_Low);
    return;
```

## Library pseudocode for shared/trace/TraceInstrumentationAllowed/TraceInstrumentationAllowed

```
// TraceInstrumentationAllowed()
// =====
// Returns TRUE if Instrumentation Trace is allowed
// in the given Exception level and Security state.

boolean TraceInstrumentationAllowed(SecurityState ss, bits(2) el)
    if !IsFeatureImplemented(FEAT_ITE) then return FALSE;
    if ELUsingAArch32(el) then return FALSE;

    if TraceAllowed(el) then
        bit ite_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;

    if SelfHostedTraceEnabled() then
        return ite_bit == '1';
    else
        bit el_bit;
        bit ss_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;
        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';

        constant boolean ed_allowed = ss_bit == '1' && el_bit == '1';

        if TRCCONFIGR.ITO == '1' then
            return ed_allowed;
        else
            return ed_allowed && ite_bit == '1';
    else
        return FALSE;
```

## Library pseudocode for shared/trace/TraceProcessElements/TraceUnitFlush

```
// TraceUnitFlush()
// =====
// Called when a trace unit flush is requested, to output previous recorded trace.

TraceUnitFlush();
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0HTRE

```
// EffectiveE0HTRE()
// =====
// Returns effective E0HTRE value

bit EffectiveE0HTRE()
    return if ELUsingAArch32(EL2) then HTRFCR.E0HTRE else TRFCR_EL2.E0HTRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE0TRE

```
// EffectiveE0TRE()
// =====
// Returns effective E0TRE value

bit EffectiveE0TRE()
    return if ELUsingAArch32(EL1) then TRFCR.E0TRE else TRFCR_EL1.E0TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE1TRE

```
// EffectiveE1TRE()
// =====
// Returns effective E1TRE value

bit EffectiveE1TRE()
    return if UsingAArch32() then TRFCR.E1TRE else TRFCR_EL1.E1TRE;
```

## Library pseudocode for shared/trace/selfhosted/EffectiveE2TRE

```
// EffectiveE2TRE()
// =====
// Returns effective E2TRE value

bit EffectiveE2TRE()
    return if UsingAArch32() then HTRFCR.E2TRE else TRFCR_EL2.E2TRE;
```

## Library pseudocode for shared/trace/selfhosted/SelfHostedTraceEnabled

```
// SelfHostedTraceEnabled()
// =====
// Returns TRUE if Self-hosted Trace is enabled.

boolean SelfHostedTraceEnabled()
    bit secure_trace_enable = '0';
    if !(HaveTraceExt() && IsFeatureImplemented(FEAT_TRF)) then return FALSE;
    if EDSCR.TFO == '0' then return TRUE;
    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()));
    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';

    if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled() then
        return TRUE;

    return FALSE;
```

## Library pseudocode for shared/trace/selfhosted/TraceAllowed

```
// TraceAllowed()
// =====
// Returns TRUE if Self-hosted Trace is allowed in the given Exception level.

boolean TraceAllowed(bits(2) el)
    if !HaveTraceExt() then
        return FALSE;
    // If in Debug state then tracing is not allowed
    if Halted() && !Restarting() then
        return FALSE;
    if SelfHostedTraceEnabled() then
        boolean trace_allowed;
        ss = 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
                bit trace_bit;
                if HaveEL(EL3) then
                    trace_bit = if ELUsingAArch32(EL3) then SDCR.STE else MDCR_EL3.STE;
                else
                    trace_bit = '1';
                trace_allowed = trace_bit == '1';
            when SS_Realm
                trace_allowed = MDCR_EL3.RLTE == '1';
            when SS_Root
                trace_allowed = FALSE;

        // 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
            (owning_ss, owning_el) = TraceBufferOwner();
            if (ss != owning_ss || UInt(owning_el) < UInt(el) ||
                (EffectiveTGE() == '1' && owning_el == EL1)) then
                trace_allowed = FALSE;

    bit TRE_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 = EffectiveE0HTRE();
            else
                TRE_bit = EffectiveE0TRE();

    return trace_allowed && TRE_bit == '1';
else
    return ExternalNoninvasiveDebugAllowed(el);
```

## Library pseudocode for shared/trace/selfhosted/TraceContextIDR2

```
// TraceContextIDR2()
// =====

boolean TraceContextIDR2()
    if !TraceAllowed(PSTATE.EL) || !HaveEL(EL2) then return FALSE;
    return (!SelfHostedTraceEnabled() || TRFCR_EL2.CX == '1');
```

## 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.

TraceSynchronizationBarrier()
    if IsFeatureImplemented(FEAT_TRBE) && !TraceAllowed(PSTATE.EL) then
        TraceUnitFlush();
    return;
```

## Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

TimeStamp TraceTimeStamp()
    if SelfHostedTraceEnabled() then
        if HaveEL(EL2) then
            TS_el2 = TRFCR_EL2.TS;
            if !IsFeatureImplemented(FEAT_ECV) && TS_el2 == '10' then
                // Reserved value
                (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP, 2);

            case TS_el2 of
                when '00'
                    // Falls out to check TRFCR_EL1.TS
                when '01'
                    return TimeStamp_Virtual;
                when '10'
                    // Otherwise ConstrainUnpredictableBits removes this case
                    assert IsFeatureImplemented(FEAT_ECV);
                    return TimeStamp_OffsetPhysical;
                when '11'
                    return TimeStamp_Physical;

            TS_el1 = TRFCR_EL1.TS;
            if TS_el1 == '00' || (!IsFeatureImplemented(FEAT_ECV) && TS_el1 == '10') then
                // Reserved value
                (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP, 2);

            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
            else
                return TimeStamp_CoreSight;
```

## Library pseudocode for shared/trace/system/IsTraceCorePowered

```
// IsTraceCorePowered()
// =====
// Returns TRUE if the Trace Core Power Domain is powered up

boolean IsTraceCorePowered();
```



## Library pseudocode for shared/translation/at

```
enumeration TranslationStage {
    TranslationStage_1,
    TranslationStage_12
};

enumeration ATAccess {
    ATAccess_Read,
    ATAccess_Write,
    ATAccess_Any,
    ATAccess_ReadPAN,
    ATAccess_WritePAN
};
```

## Library pseudocode for shared/translation/at/EncodePARAttr

```
// EncodePARAttr()
// =====
// Convert orthogonal attributes and hints to 64-bit PAR ATTR field.

bits(8) EncodePARAttr(MemoryAttributes memattr)
    bits(8) result;

    if IsFeatureImplemented(FEAT_MTE) && memattr.tags == MemTag\_AllocationTagged then
        if IsFeatureImplemented(FEAT_MTE_PERM) && memattr.notagaccess then
            result<7:0> = '11100000';
        else
            result<7:0> = '11110000';
        return result;

    if memattr.memtype == MemType\_Device then
        result<7:4> = '0000';
        case memattr.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();
        result<0> = NOT memattr.xs;
    else
        if memattr.xs == '0' then
            if (memattr.outer.attrs == MemAttr\_WT && memattr.inner.attrs == MemAttr\_WT &&
                !memattr.outer.transient && memattr.outer.hints == MemHint\_RA) then
                return '10100000';
            elsif memattr.outer.attrs == MemAttr\_NC && memattr.inner.attrs == MemAttr\_NC then
                return '01000000';

            if memattr.outer.attrs == MemAttr\_WT then
                result<7:6> = if memattr.outer.transient then '00' else '10';
                result<5:4> = memattr.outer.hints;
            elsif memattr.outer.attrs == MemAttr\_WB then
                result<7:6> = if memattr.outer.transient then '01' else '11';
                result<5:4> = memattr.outer.hints;
            else // MemAttr_NC
                result<7:4> = '0100';

            if memattr.inner.attrs == MemAttr\_WT then
                result<3:2> = if memattr.inner.transient then '00' else '10';
                result<1:0> = memattr.inner.hints;
            elsif memattr.inner.attrs == MemAttr\_WB then
                result<3:2> = if memattr.inner.transient then '01' else '11';
                result<1:0> = memattr.inner.hints;
            else // MemAttr_NC
                result<3:0> = '0100';

        return result;
```

## Library pseudocode for shared/translation/at/PAREncodeShareability

```
// PAREncodeShareability()
// =====
// Derive 64-bit PAR SH field.

bits(2) PAREncodeShareability(MemoryAttributes memattrs)
    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';

    case memattrs.shareability of
        when Shareability\_NSH return '00';
        when Shareability\_ISH return '11';
        when Shareability\_OSH return '10';
```

## 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.

bits(8) ReportedPARAttrs(bits(8) parattrs);
```

## 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.

bits(2) ReportedPARShareability(bits(2) sh);
```

## Library pseudocode for shared/translation/attrs/DecodeDevice

```
// DecodeDevice()
// =====
// Decode output Device type

DeviceType DecodeDevice(bits(2) device)
    case device of
        when '00' return DeviceType\_nGnRnE;
        when '01' return DeviceType\_nGnRE;
        when '10' return DeviceType\_nGRE;
        when '11' return DeviceType\_GRE;
```

## Library pseudocode for shared/translation/attrs/DecodeLDFAttr

```
// DecodeLDFAttr()
// =====
// Decode memory attributes using LDF (Long Descriptor Format) mapping

MemAttrHints DecodeLDFAttr(bits(4) attr)
    MemAttrHints ldfattr;

    if attr == 'x0xx' then ldfattr.attrs = MemAttr_WT; // Write-through
    elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
    elsif attr == 'x1xx' then ldfattr.attrs = MemAttr_WB; // Write-back
    else
        Unreachable();

    // Allocation hints are applicable only to cacheable memory.
    if ldfattr.attrs != MemAttr_NC then
        case attr<1:0> of
            when '00' ldfattr.hints = MemHint_No; // No allocation hints
            when '01' ldfattr.hints = MemHint_WA; // Write-allocate
            when '10' ldfattr.hints = MemHint_RA; // Read-allocate
            when '11' ldfattr.hints = MemHint_RWA; // Read/Write allocate

    // The Transient hint applies only to cacheable memory with some allocation hints.
    if ldfattr.attrs != MemAttr_NC && ldfattr.hints != MemHint_No then
        ldfattr.transient = attr<3> == '0';

    return ldfattr;
```

## Library pseudocode for shared/translation/attrs/DecodeSDFAttr

```
// DecodeSDFAttr()
// =====
// Decode memory attributes using SDF (Short Descriptor Format) mapping

MemAttrHints DecodeSDFAttr(bits(2) rgn)
    MemAttrHints sdfattr;

    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;

    sdfattr.transient = FALSE;

    return sdfattr;
```

## Library pseudocode for shared/translation/attrs/DecodeShareability

```
// DecodeShareability()
// =====
// Decode shareability of target memory region

Shareability DecodeShareability(bits(2) sh)
    case sh of
        when '10' return Shareability\_OSH;
        when '11' return Shareability\_ISH;
        when '00' return Shareability\_NSH;
        otherwise
            case ConstrainUnpredictable(Unpredictable Shareability) of
                when Constraint\_OSH return Shareability\_OSH;
                when Constraint\_ISH return Shareability\_ISH;
                when Constraint\_NSH return Shareability\_NSH;
```

## Library pseudocode for shared/translation/attrs/EffectiveShareability

```
// EffectiveShareability()
// =====
// Force Outer Shareability on Device and Normal iNCoNC memory

Shareability EffectiveShareability(MemoryAttributes memattrs)
    if (memattrs.memtype == MemType\_Device ||
        (memattrs.inner.attrs == MemAttr\_NC &&
         memattrs.outer.attrs == MemAttr\_NC)) then
        return Shareability\_OSH;
    else
        return memattrs.shareability;
```

## Library pseudocode for shared/translation/attrs/IsWBShareable

```
// IsWBShareable()
// =====
// Determines whether the given memory attributes are iWBoWB Shareable

boolean IsWBShareable(MemoryAttributes memattrs)

    return (memattrs.memtype == MemType\_Normal &&
            memattrs.inner.attrs == MemAttr\_WB &&
            memattrs.outer.attrs == MemAttr\_WB &&
            memattrs.shareability IN {Shareability\_ISH, Shareability\_OSH});
```

## Library pseudocode for shared/translation/attrs/NormalNCMemAttr

```
// NormalNCMemAttr()
// =====
// Normal Non-cacheable memory attributes

MemoryAttributes NormalNCMemAttr()
    MemAttrHints non_cacheable;
    non_cacheable.attrs = MemAttr\_NC;

    MemoryAttributes nc_memattrs;
    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;
```

## Library pseudocode for shared/translation/attrs/S1ConstrainUnpredictableRESMAIR

```
// S1ConstrainUnpredictableRESMAIR()
// =====
// Determine whether a reserved value occupies MAIR_ELx.AttrN

boolean S1ConstrainUnpredictableRESMAIR(bits(8) attr, boolean slaarch64)
  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;
```

## Library pseudocode for shared/translation/attrs/S1DecodeMemAttrs

```
// S1DecodeMemAttrs()
// =====
// Decode MAIR-format memory attributes assigned in stage 1

MemoryAttributes S1DecodeMemAttrs(bits(8) attr_in, bits(2) sh, boolean slaarch64,
                                   S1TWPParams walkparams)

bits(8) attr = attr_in;
if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
    // Map reserved encodings to an allocated encoding
    (-, attr) = ConstrainUnpredictableBits(Unpredictable\_RESMAIR, 8);

MemoryAttributes memattrs;
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';
if slaarch64 && attr == '11110000' then
    memattrs.tags = MemTag\_AllocationTagged;
elsif slaarch64 && walkparams.mtx == '1' then
    memattrs.tags = MemTag\_CanonicallyTagged;
else
    memattrs.tags = MemTag\_Untagged;

memattrs.notagaccess = FALSE;

memattrs.shareability = DecodeShareability(sh);

return memattrs;
```

## 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.

MemAttrHints S2CombineS1AttrHints(MemAttrHints s1_attrhints, MemAttrHints s2_attrhints)
    MemAttrHints attrhints;

    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;

    // 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;

    return attrhints;
```

## 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

DeviceType S2CombineS1Device(DeviceType s1_device, DeviceType s2_device)
    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;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1MemAttrs

```
// S2CombineS1MemAttrs()
// =====
// Combine stage 2 with stage 1 memory attributes

MemoryAttributes S2CombineS1MemAttrs(MemoryAttributes s1_memattrs, MemoryAttributes s2_memattrs,
                                     boolean s2aarch64)

    MemoryAttributes memattrs;

    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);
    elseif s1_memattrs.memtype == MemType_Device then // S2 Normal, S1 Device
        memattrs = s1_memattrs;
    elseif 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);

    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);
    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';
    elseif s2aarch64 then
        memattrs.xs = s2_memattrs.xs AND s1_memattrs.xs;
    else
        memattrs.xs = s1_memattrs.xs;

    memattrs.shareability = EffectiveShareability(memattrs);
    return memattrs;
```

## Library pseudocode for shared/translation/attrs/S2CombineS1Shareability

```
// S2CombineS1Shareability()
// =====
// Combine stage 2 shareability with stage 1

Shareability S2CombineS1Shareability(Shareability s1_shareability,
                                     Shareability s2_shareability)

    if (s1_shareability == Shareability_OSH ||
        s2_shareability == Shareability_OSH) then
        return Shareability_OSH;
    elseif (s1_shareability == Shareability_ISH ||
           s2_shareability == Shareability_ISH) then
        return Shareability_ISH;
    else
        return Shareability_NSH;
```



## Library pseudocode for shared/translation/attrs/S2DecodeCacheability

```
// S2DecodeCacheability()
// =====
// Determine the stage 2 cacheability for Normal memory

MemAttrHints S2DecodeCacheability(bits(2) attr)
    MemAttrHints s2attr;

    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();

    // 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      = bits(2) UNKNOWN;
    s2attr.transient = boolean UNKNOWN;

    return s2attr;
```

## Library pseudocode for shared/translation/attrs/S2DecodeMemAttrs

```
// S2DecodeMemAttrs()
// =====
// Decode stage 2 memory attributes when FWB is 0

MemoryAttributes S2DecodeMemAttrs(bits(4) attr_in, bits(2) sh, boolean s2aarch64)
    MemoryAttributes memattrs;

    bits(4) attr;
    if S2ResMemAttr(s2aarch64, attr_in) then
        // Map reserved encodings to an allocated encoding
        (-, attr) = ConstrainUnpredictableBits(Unpredictable\_S2RESMEMATTR, 4);
    else
        attr = attr_in;

    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>);

    memattrs.shareability = DecodeShareability(sh);

    if s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM) then
        memattrs.notagaccess = attr == '0100';
    else
        memattrs.notagaccess = FALSE;

    return memattrs;
```

## Library pseudocode for shared/translation/attrs/S2MemTagType

```
// S2MemTagType()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

MemTagType S2MemTagType(MemoryAttributes s2_memattrs, MemTagType s1_tagtype)

    if !IsFeatureImplemented(FEAT_MTE2) then
        return MemTag_Untagged;

    if ((s1_tagtype == MemTag_AllocationTagged) &&
        (s2_memattrs.memtype == MemType_Normal) &&
        (s2_memattrs.inner.attrs == MemAttr_WB) &&
        (s2_memattrs.inner.hints == MemHint_RWA) &&
        (!s2_memattrs.inner.transient) &&
        (s2_memattrs.outer.attrs == MemAttr_WB) &&
        (s2_memattrs.outer.hints == MemHint_RWA) &&
        (!s2_memattrs.outer.transient)) then
        return MemTag_AllocationTagged;

    // Return what stage 1 asked for if we can, otherwise Untagged.
    if s1_tagtype != MemTag_AllocationTagged then
        return s1_tagtype;

    return MemTag_Untagged;
```

## Library pseudocode for shared/translation/attrs/S2ResMemAttr

```
// S2ResMemAttr()
// =====
// Determine whether a reserved value occupies stage 2 MemAttr field when FWB is 0

boolean S2ResMemAttr(boolean s2aarch64, bits(4) attr)
    case attr of
        when '0100' return !(s2aarch64 && IsFeatureImplemented(FEAT_MTE_PERM));
        when '1x00' return TRUE;
        otherwise return FALSE;
```

## Library pseudocode for shared/translation/attrs/WalkMemAttrs

```
// WalkMemAttrs()
// =====
// Retrieve memory attributes of translation table walk

MemoryAttributes WalkMemAttrs(bits(2) sh, bits(2) irgn, bits(2) orgn)
    MemoryAttributes walkmemattrs;

    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';
    walkmemattrs.notagaccess = FALSE;

    return walkmemattrs;
```

## 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

FaultRecord AlignmentFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault;

    fault.statuscode = Fault\_Alignment;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();
    fault.vaddress = vaddress;

    return fault;
```

## Library pseudocode for shared/translation/faults/ExclusiveFault

```
// ExclusiveFault()
// =====
// Return a fault record indicating an Exclusive fault for a specific access

FaultRecord ExclusiveFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault;

    fault.statuscode = Fault\_Exclusive;
    fault.accessdesc = accdesc;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;
    fault.write = !accdesc.read && accdesc.write;
    fault.gpcfs2walk = FALSE;
    fault.gpcf = GPCNoFault();
    fault.vaddress = vaddress;

    return fault;
```

## Library pseudocode for shared/translation/faults/NoFault

```
// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred

FaultRecord NoFault()
    FaultRecord fault;

    fault.vaddress = bits(64) UNKNOWN;
    fault.statuscode = Fault None;
    fault.accessdesc = AccessDescriptor UNKNOWN;
    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;

// NoFault()
// =====
// Return a clear fault record indicating no faults have occurred for a specific access

FaultRecord NoFault(AccessDescriptor accdesc)
    FaultRecord fault = NoFault();
    fault.accessdesc = accdesc;
    fault.write = !accdesc.read && accdesc.write;

    return fault;

// NoFault()
// =====

FaultRecord NoFault(AccessDescriptor accdesc, bits(64) vaddress)
    FaultRecord fault = NoFault();
    fault.accessdesc = accdesc;
    fault.write = !accdesc.read && accdesc.write;
    fault.vaddress = vaddress;

    return fault;
```

## Library pseudocode for shared/translation/gpc/AbovePPS

```
// AbovePPS()
// =====
// Returns TRUE if an address exceeds the range configured in GPCCR_EL3.PPS.

boolean AbovePPS(bits(56) address)
    constant integer pps = DecodePPS();
    if pps >= 56 then
        return FALSE;

    return IsZero(address<55:pps>);
```

## Library pseudocode for shared/translation/gpc/DecodeGPTBlock

```
// DecodeGPTBlock()
// =====
// Decode a GPT Block descriptor.

GPTEntry DecodeGPTBlock(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Block;
    GPTEntry result;
    result.gpi = gpt_entry<7:4>;
    result.level = 0;

    // 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();
    result.contig_size = GPTL0Size();

    return result;
```

## Library pseudocode for shared/translation/gpc/DecodeGPTContiguous

```
// DecodeGPTContiguous()
// =====
// Decode a GPT Contiguous descriptor.

GPTEntry DecodeGPTContiguous(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Contig;
    GPTEntry result;
    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();

    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();

    result.level = 1;

    return result;
```

## Library pseudocode for shared/translation/gpc/DecodeGPTGranules

```
// DecodeGPTGranules()
// =====
// Decode a GPT Granules descriptor.

GPTEntry DecodeGPTGranules(PGSe pgs, integer index, bits(64) gpt_entry)
    GPTEntry result;
    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();

    result.contig_size = result.size; // No contiguity
    result.level = 1;

    return result;
```

## Library pseudocode for shared/translation/gpc/DecodeGPTTable

```
// DecodeGPTTable()
// =====
// Decode a GPT Table descriptor.

GPTTable DecodeGPTTable(PGSe pgs, bits(64) gpt_entry)
    assert gpt_entry<3:0> == GPT_Table;
    GPTTable result;

    case pgs of
        when PGS_4KB    result.address = gpt_entry<55:17>:Zeros(17);
        when PGS_16KB   result.address = gpt_entry<55:15>:Zeros(15);
        when PGS_64KB   result.address = gpt_entry<55:13>:Zeros(13);
        otherwise Unreachable();

    return result;
```

## Library pseudocode for shared/translation/gpc/DecodePGS

```
// DecodePGS()
// =====

PGSe DecodePGS(bits(2) pgs)
    case pgs of
        when '00' return PGS_4KB;
        when '10' return PGS_16KB;
        when '01' return PGS_64KB;
        otherwise Unreachable();
```

## Library pseudocode for shared/translation/gpc/DecodePGSRange

```
// DecodePGSRange()
// =====

AddressSize DecodePGSRange(PGSe pgs)
    case pgs of
        when PGS_4KB    return GPTRange_4KB;
        when PGS_16KB   return GPTRange_16KB;
        when PGS_64KB   return GPTRange_64KB;
        otherwise Unreachable();
```

## Library pseudocode for shared/translation/gpc/DecodePPS

```
// DecodePPS()
// =====
// Size of region protected by the GPT, in bits.

AddressSize DecodePPS()
    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();
```

## Library pseudocode for shared/translation/gpc/GPCBW\_EL3BWSTRIDEValid

```
// GPCBW_EL3BWSTRIDEValid()
// =====
// Returns whether the current GPCBW_EL3.BWSTRIDE value is valid

boolean GPCBW_EL3BWSTRIDEValid()
    assert IsFeatureImplemented(FEAT_RME_GPC3);
    return GPCBW_EL3.BWSTRIDE IN {
        '00000',
        '00010',
        '00100',
        '00110',
        '00111',
        '01000',
        '01001',
        '01010',
        '10000'
    };
```

## Library pseudocode for shared/translation/gpc/GPCFault

```
// GPCFault()
// =====
// Constructs and returns a GPCF

GPCFRecord GPCFault(GPCF gpf, integer level)
    GPCFRecord fault;
    fault.gpf = gpf;
    fault.level = level;
    return fault;
```

## Library pseudocode for shared/translation/gpc/GPCNoFault

```
// GPCNoFault()
// =====
// Returns the default properties of a GPCF that does not represent a fault

GPCFRecord GPCNoFault()
    GPCFRecord result;
    result.gpf = GPCF\_None;
    return result;
```

## 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.

boolean GPCRegistersConsistent()
    // Check for Reserved register values
    if IsFeatureImplemented(FEAT_RME_GPC3) then
        if ! GPCCR_EL3.<PPS3, PPS> IN {'0xxx', '100x'} then
            return FALSE;

        if GPCCR_EL3.GPCBW == '1' then
            if ! GPCBW_EL3.BWSIZE IN {'00x', '1x0', '010'} then
                return FALSE;

            if !GPCBW\_EL3BWSTRIDEValid() then
                return FALSE;

            if !IsAligned(GPCBW_EL3.BWADDR, 1 << UInt(GPCBW_EL3.BWSIZE)) then
                return FALSE;
    else
        if GPCCR_EL3.PPS == '111' then
            return FALSE;

        if DecodePPS() > AArch64.PAMax() then
            return FALSE;
        if GPCCR_EL3.PGS == '11' then
            return FALSE;
        if GPCCR_EL3.SH == '01' then
            return FALSE;

    // Inner and Outer Non-cacheable requires Outer Shareable
    if GPCCR_EL3.<ORGN, IRGN> == '0000' && GPCCR_EL3.SH != '10' then
        return FALSE;

    return TRUE;
```



## 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

boolean GPICheck(PASpace paspace, bits(4) gpi, SecurityState ss)
    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();
```

## Library pseudocode for shared/translation/gpc/GPIIndex

```
// GPIIndex()
// =====

integer GPIIndex(bits(56) pa)
    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();
```

## Library pseudocode for shared/translation/gpc/GPIValid

```
// GPIValid()
// =====
// Returns whether a given value is a valid encoding for a GPI value

boolean GPIValid(bits(4) gpi)
    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;
```

## Library pseudocode for shared/translation/gpc/GPT

```
// GPT dimensions
// =====

constant AddressSize GPTRange_4KB    = 12;
constant AddressSize GPTRange_16KB   = 14;
constant AddressSize GPTRange_64KB   = 16;
constant AddressSize GPTRange_2MB    = 21;
constant AddressSize GPTRange_32MB   = 25;
constant AddressSize GPTRange_512MB  = 29;
constant AddressSize GPTRange_1GB    = 30;
constant AddressSize GPTRange_16GB   = 34;
constant AddressSize GPTRange_64GB   = 36;
constant AddressSize GPTRange_512GB  = 39;
```

## Library pseudocode for shared/translation/gpc/GPTBlockDescriptorValid

```
// GPTBlockDescriptorValid()
// =====
// Returns TRUE if the given GPT Block descriptor is valid, and FALSE otherwise.

boolean GPTBlockDescriptorValid(bits(64) level_0_entry)
    assert level_0_entry<3:0> == GPT\_Block;
    return IsZero(level_0_entry<63:8>) && GPIValid(level_0_entry<7:4>);
```

## Library pseudocode for shared/translation/gpc/GPTContigDescriptorValid

```
// GPTContigDescriptorValid()
// =====
// Returns TRUE if the given GPT Contiguous descriptor is valid, and FALSE otherwise.

boolean GPTContigDescriptorValid(bits(64) level_1_entry)
    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>));
```

## Library pseudocode for shared/translation/gpc/GPTEntry

```
// GPTEntry
// =====

type GPTEntry is (
    bits(4) gpi,           // GPI value for this region
    AddressSize size,      // Region size
    AddressSize contig_size, // Contiguous region size
    integer level,        // Level of GPT lookup
    bits(56) pa           // 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.

boolean GPTGranulesDescriptorValid(bits(64) level_1_entry)
    for i = 0 to 15
        if !GPIValid(level_1_entry<i*4 +:4>) then
            return FALSE;
    return TRUE;
```

## Library pseudocode for shared/translation/gpc/GPTL0Size

```
// GPTL0Size()
// =====
// Returns number of bits covered by a level 0 GPT entry

AddressSize GPTL0Size()
    case GPCCR_EL3.L0GPTSZ of
        when '0000' return GPTRange\_1GB;
        when '0100' return GPTRange\_16GB;
        when '0110' return GPTRange\_64GB;
        when '1001' return GPTRange\_512GB;
        otherwise Unreachable();
    return 30;
```

## Library pseudocode for shared/translation/gpc/GPTLevel0EntryValid

```
// GPTLevel0EntryValid()
// =====
// Returns TRUE if the given level 0 gpt descriptor is valid, and FALSE otherwise.

boolean GPTLevel0EntryValid(bits(64) gpt_entry)
    case gpt_entry<3:0> of
        when GPT\_Block return GPTBlockDescriptorValid(gpt_entry);
        when GPT\_Table return GPTTableDescriptorValid(gpt_entry);
        otherwise return FALSE;
```

## Library pseudocode for shared/translation/gpc/GPTLevel0Index

```
// GPTLevel0Index()
// =====
// Compute the level 0 index based on input PA.

integer GPTLevel0Index(bits(56) pa)
    // Input address and index bounds
    constant integer pps = DecodePPS();
    constant integer l0sz = GPTL0Size();
    if pps <= l0sz then
        return 0;

    return UInt(pa<pps-1:l0sz>);
```

## Library pseudocode for shared/translation/gpc/GPTLevel1EntryValid

```
// GPTLevel1EntryValid()
// =====
// Returns TRUE if the given level 1 gpt descriptor is valid, and FALSE otherwise.

boolean GPTLevel1EntryValid(bits(64) gpt_entry)
    case gpt_entry<3:0> of
        when GPT\_Contig return GPTContigDescriptorValid(gpt_entry);
        otherwise      return GPTGranulesDescriptorValid(gpt_entry);
```

## Library pseudocode for shared/translation/gpc/GPTLevel1Index

```
// GPTLevel1Index()
// =====
// Compute the level 1 index based on input PA.

integer GPTLevel1Index(bits(56) pa)
    // Input address and index bounds
    constant integer l0sz = 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();
```

## Library pseudocode for shared/translation/gpc/GPTTable

```
// GPTTable
// =====

type GPTTable is (
    bits(56) address          // 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.

boolean GPTTableDescriptorValid(bits(64) level_0_entry)
    assert level_0_entry<3:0> == GPT\_Table;
    constant integer l0sz = GPTL0Size();
    constant PGSe pgs      = DecodePGS(GPCCR_EL3.PGS);
    constant integer p      = DecodePGSRange(pgs);
    return IsZero(level_0_entry<63:52,11:4>) && IsZero(level_0_entry<(l0sz-p)-2:12>);
```



```

// GPTWalk()
// =====
// Get the GPT entry for a given physical address, pa

(GPCFRecord, GPTEntry) GPTWalk(bits(56) pa, AccessDescriptor accdesc)
    // GPT base address
    bits(56) base;
    pgs = 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, LOGPSSZ}.
    // * 4KB
    base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    pps = DecodePPS();
    l0sz = GPTL0Size();
    constant integer alignment = Max((pps - l0sz) + 3, 12);
    base<alignment-1:0> = Zeros(alignment);

    constant AccessDescriptor gptaccdesc = CreateAccDescGPTW(accdesc);

    // Access attributes and address for GPT fetches
    AddressDescriptor gptaddrdesc;
    gptaddrdesc.memattrs = WalkMemAttrs(GPCCR_EL3.SH, GPCCR_EL3.ORG, GPCCR_EL3.IRG);
    gptaddrdesc.fault = NoFault(gptaccdesc);

    gptaddrdesc.paddress.paspace = PAS\_Root;
    gptaddrdesc.paddress.address = base + GPTLevel0Index(pa) * 8;

    // Fetch LOGPT entry
    bits(64) level_0_entry;
    PhysMemRetStatus memstatus;
    (memstatus, level_0_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF\_EABT, 0), GPTEntry UNKNOWN);

    if !GPTLevel0EntryValid(level_0_entry) then
        return (GPCFault(GPCF\_Walk, 0), GPTEntry UNKNOWN);

    GPTEntry result;
    GPTTable table;
    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), GPTEntry UNKNOWN);
            otherwise
                // An invalid encoding would be caught by GPTLevel0EntryValid()
                Unreachable();

    // Must be a GPT Table entry
    assert level_0_entry<3:0> == GPT\_Table;

    // Address of level 1 GPT entry
    offset = GPTLevel1Index(pa) * 8;

    bits(64) level_1_entry;

    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 = Align(table.address + offset, 16);
        bits(64) level_1_entry_lower;
        (memstatus, level_1_entry_lower) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);

```

```

if IsFault(memstatus) then
    return (GPCFault(GPCF\_EABT, 1), GPTEntry UNKNOWN);

gptaddrdesc.paddress.address = gptaddrdesc.paddress.address + 8;
bits(64) level_1_entry_upper;
(memstatus, level_1_entry_upper) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
if IsFault(memstatus) then
    return (GPCFault(GPCF\_EABT, 1), GPTEntry UNKNOWN);

// 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), GPTEntry UNKNOWN);

if offset<3> == '1' then
    level_1_entry = level_1_entry_upper;
else
    level_1_entry = level_1_entry_lower;
else
    gptaddrdesc.paddress.address = table.address + offset;
    // Fetch L1GPT entry
    (memstatus, level_1_entry) = PhysMemRead(gptaddrdesc, 8, gptaccdesc);
    if IsFault(memstatus) then
        return (GPCFault(GPCF\_EABT, 1), GPTEntry UNKNOWN);

    if !GPTLevel1EntryValid(level_1_entry) then
        return (GPCFault(GPCF\_Walk, 1), GPTEntry UNKNOWN);

case level_1_entry<3:0> of
    when GPT\_Contig
        result = DecodeGPTContiguous(pgs, level_1_entry);
    otherwise
        gpi_index = GPIIndex(pa);
        result = DecodeGPTGranules(pgs, gpi_index, level_1_entry);

result.pa = pa;
return (GPCNoFault(), result);

```

## 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.

GPCFRecord GranuleProtectionCheck(AddressDescriptor addrdesc, AccessDescriptor accdesc)

    assert IsFeatureImplemented(FEAT_RME);
    // The address to be checked
    address = addrdesc.paddress;

    // Bypass mode - all accesses pass
    if GPCCR_EL3.GPC == '0' then
        return GPCNoFault();

    // Configuration consistency check
    if !GPCRegistersConsistent() then
        return GPCFault(GPCF\_Walk, 0);

    if IsFeatureImplemented(FEAT_RME_GPC2) then
        boolean access_disabled;

        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();

        if access_disabled then
            return GPCFault(GPCF\_Fail, 0);

    // 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);

    if (IsFeatureImplemented(FEAT_RME_GPC3) && GPCCR_EL3.GPCBW == '1' &&
        PAWithinGPBypassWindow(address.address)) then
        return GPCNoFault();

    // GPT base address size check
    constant bits(56) gpt_base = ZeroExtend(GPTBR_EL3.BADDR:Zeros(12), 56);
    if AbovePPS(gpt_base) then
        return GPCFault(GPCF\_AddressSize, 0);

    // GPT lookup
    (gpcf, gpt_entry) = GPTWalk(address.address, accdesc);
    if gpcf.gpf != GPCF\_None then
        return gpcf;

    // Check input physical address space against GPI
    permitted = GPICheck(address.paspace, gpt_entry.gpi, accdesc.ss);

    if !permitted then
        gpcf = GPCFault(GPCF\_Fail, gpt_entry.level);
        return gpcf;

    // Check passed

    return GPCNoFault();
```



## Library pseudocode for shared/translation/gpc/PAWithinGPGBypassWindow

```
// PAWithinGPGBypassWindow()
// =====
// Check if the supplied address is within a GPC Bypass window.

boolean PAWithinGPGBypassWindow(bits(56) pa_in)
    // Only check the top 26 bits as the minimum window size is 1GB
    constant bits(26) pa = pa_in<55:30>;

    constant integer gpcbw1 = UInt(GPCBW_EL3.BWSIZE);
    constant integer gpcbwu = 9 + UInt(GPCBW_EL3.BWSTRIDE);

    return pa<gpcbwu:gpcbw1> == GPCBW_EL3.BWADDR<gpcbwu:gpcbw1>;
```

## Library pseudocode for shared/translation/gpc/PGS

```
// PGS
// ===
// Physical granule size

enumeration PGSe {
    PGS_4KB,
    PGS_16KB,
    PGS_64KB
};
```

## Library pseudocode for shared/translation/gpc/Table

```
// Table format information
// =====
// Granule Protection Table constants

constant bits(4) GPT_NoAccess      = '0000';
constant bits(4) GPT_Table        = '0011';
constant bits(4) GPT_Block        = '0001';
constant bits(4) GPT_Contig       = '0001';
constant bits(4) GPT_SystemAgent  = '0100';
constant bits(4) GPT_NonSecureProtected = '0101';
constant bits(4) GPT_NA6          = '0110';
constant bits(4) GPT_NA7          = '0111';
constant bits(4) GPT_Secure       = '1000';
constant bits(4) GPT_NonSecure    = '1001';
constant bits(4) GPT_Root         = '1010';
constant bits(4) GPT_Realm        = '1011';
constant bits(4) GPT_NonSecureOnly = '1101';
constant bits(4) GPT_Any          = '1111';
```

## Library pseudocode for shared/translation/translation/S1TranslationRegime

```
// S1TranslationRegime()
// =====
// Stage 1 translation regime for the given Exception level

bits(2) S1TranslationRegime(bits(2) el)
    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;

// 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.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

## Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
constant integer FINAL_LEVEL = 3;

// AddressDescriptor
// =====
// Descriptor used to access the underlying memory array.

type AddressDescriptor is (
    FaultRecord      fault,          // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress      paddress,
    boolean          slassured,     // Stage 1 Assured Translation Property
    boolean          s2fslmro,     // Stage 2 MRO permission for Stage 1
    bits(16)         mecid,        // FEAT_MEC: Memory Encryption Context ID
    bits(64)         vaddress
)
```

## Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(bit d128, TGx tgx, integer level)
    if d128 == '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;
            when TGx\_64KB
                assert level IN {2, 3};
                return if level == 2 then 6 else 4;
        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;
```

## Library pseudocode for shared/translation/vmsa/CreateAddressDescriptor

```
// CreateAddressDescriptor()
// =====
// Set internal members for address descriptor type to valid values

AddressDescriptor CreateAddressDescriptor(bits(64) va, FullAddress pa,
                                         MemoryAttributes memattrs)

    AddressDescriptor addrdesc;

    addrdesc.paddress = pa;
    addrdesc.vaddress = va;
    addrdesc.memattrs = memattrs;
    addrdesc.fault    = NoFault();
    addrdesc.slassured = FALSE;

    return addrdesc;
```

## Library pseudocode for shared/translation/vmsa/CreateFaultyAddressDescriptor

```
// CreateFaultyAddressDescriptor()
// =====
// Set internal members for address descriptor type with values indicating error

AddressDescriptor CreateFaultyAddressDescriptor(bits(64) va, FaultRecord fault)
    AddressDescriptor addrdesc;

    addrdesc.vaddress = va;
    addrdesc.fault    = fault;

    return addrdesc;
```

## Library pseudocode for shared/translation/vmsa/DecodePASpace

```
// DecodePASpace()
// =====
// Decode the target PA Space

PASpace DecodePASpace(bit nse2, bit nse, bit ns)
    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();
```

## Library pseudocode for shared/translation/vmsa/DescriptorType

```
// DescriptorType
// =====
// Translation table descriptor formats

enumeration DescriptorType {
    DescriptorType_Table,
    DescriptorType_Leaf,
    DescriptorType_Invalid
};
```

## Library pseudocode for shared/translation/vmsa/Domains

```
// Domains
// =====
// Short-descriptor format Domains

constant bits(2) Domain_NoAccess = '00';
constant bits(2) Domain_Client   = '01';
constant bits(2) Domain_Manager  = '11';
```

## Library pseudocode for shared/translation/vmsa/FetchDescriptor

```
// FetchDescriptor()
// =====
// Fetch a translation table descriptor

(FaultRecord, bits(N)) FetchDescriptor(bit ee, AddressDescriptor walkaddress,
                                       AccessDescriptor walkaccess, FaultRecord fault_in,
                                       integer N)

// 32-bit descriptors for AArch32 Short-descriptor format
// 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
// 128-bit descriptors for AArch64 when FEAT_D128 is set and {V}TCR_ELx.d128 is set
assert N == 32 || N == 64 || N == 128;
bits(N) descriptor;
FaultRecord fault = 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, bits(N) UNKNOWN);

PhysMemRetStatus memstatus;
(memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkaccess);
if IsFault(memstatus) then
    constant boolean iswrite = FALSE;
    fault = HandleExternalTTWAbort(memstatus, iswrite, walkaddress,
                                   walkaccess, N DIV 8, fault);

    if IsFault(fault.statuscode) then
        return (fault, bits(N) UNKNOWN);

if ee == '1' then
    descriptor = BigEndianReverse(descriptor);

return (fault, descriptor);
```

## Library pseudocode for shared/translation/vmsa/HasUnprivileged

```
// HasUnprivileged()
// =====
// Returns whether a translation regime serves EL0 as well as a higher EL

boolean HasUnprivileged(Regime regime)
    return (regime IN {
        Regime\_EL20,
        Regime\_EL30,
        Regime\_EL10
    });
```

## Library pseudocode for shared/translation/vmsa/Regime

```
// Regime
// =====
// Translation regimes

enumeration Regime {
    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  
  
boolean RegimeUsingAArch32(Regime regime)  
    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;
```

## 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 is (
// A64-VMSA exclusive parameters
    bit      ha,          // TCR_ELx.HA
    bit      hd,          // TCR_ELx.HD
    bit      tbi,         // TCR_ELx.TBI{x}
    bit      tbid,        // TCR_ELx.TBID{x}
    bit      nfd,         // TCR_EL1.NFDx or TCR_EL2.NFDx when HCR_EL2.E2H == '1'
    bit      e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    bit      dl28,        // TCR_ELx.D128
    bit      aie,         // (TCR2_ELx/TCR_EL3).AIE
    MAIRType  mair2,      // MAIR2_ELx
    bit      ds,          // TCR_ELx.DS
    bits(3)   ps,         // TCR_ELx.{I}PS
    bits(6)   txsz,       // TCR_ELx.TxSZ
    bit      epan,        // SCTLR_EL1.EPAN or SCTLR_EL2.EPAN when HCR_EL2.E2H == '1'
    bit      dct,         // HCR_EL2.DCT
    bit      nv1,        // HCR_EL2.NV1
    bit      cmow,        // SCTLR_EL1.CMOW or SCTLR_EL2.CMOW when HCR_EL2.E2H == '1'
    bit      pnch,        // TCR{2}_ELx.PnCH
    bit      disch,       // TCR{2}_ELx.DisCH
    bit      haft,        // TCR{2}_ELx.HAFT
    bit      mtX,         // TCR_ELx.MTX{y}
    bits(2)   skl,        // TTBRn_ELx.SKl
    bit      pie,         // TCR2_ELx.PIE or TCR_EL3.PIE
    S1PIRType pir,        // PIR_ELx
    S1PIRType pire0,     // PIRE0_EL1 or PIRE0_EL2 when HCR_EL2.E2H == '1'
    bit      emec,        // SCTLR2_EL2.EMEC or SCTLR2_EL3.EMEC
    bit      amec,        // TCR2_EL2.AMEC0 or TCR2_EL2.AMEC1 when HCR_EL2.E2H == '1'
    bit      fng,         // TCR2_EL1.FNGx or TCR2_EL2.FNGx when HCR_EL2.E2H == '1'
    bit      fngna,       // TCR2_EL1.FNGx

// A32-VMSA exclusive parameters
    bits(3)   t0sz,       // TTBCR.T0SZ
    bits(3)   t1sz,       // TTBCR.T1SZ
    bit      uwxn,        // SCTLR.UWXN

// Parameters common to both A64-VMSA & A32-VMSA (A64/A32)
    TGx      tgX,         // TCR_ELx.TGx      / Always TGx_4KB
    bits(2)   irgn,       // TCR_ELx.IRGNx    / TTBCR.IRGNx or HTCR.IRGN0
    bits(2)   orgn,       // TCR_ELx.ORGNx    / TTBCR.ORGnx or HTCR.ORGNO
    bits(2)   sh,         // TCR_ELx.SHx      / TTBCR.SHx or HTCR.SH0
    bit      hpd,         // TCR_ELx.HPD{x}   / TTBCR2.HPDx or HTCR.HPD
    bit      ee,          // SCTLR_ELx.EE     / SCTLR.EE or HSCTLR.EE
    bit      wxn,         // SCTLR_ELx.WXN    / SCTLR.WXN or HSCTLR.WXN
    bit      ntlsmD,      // SCTLR_ELx.nTLsMD / SCTLR.nTLsMD or HSCTLR.nTLsMD
    bit      dc,          // HCR_EL2.DC       / HCR.DC
    bit      sif,         // SCR_EL3.SIF      / SCR.SIF
    MAIRType  mair        // MAIR_ELx         / MAIR1:MAIR0 or HMAIR1:HMAIR0
)
```

## Library pseudocode for shared/translation/vmsa/S2TTWParams

```
// S2TTWParams
// =====
// Register fields corresponding to stage 2 translation.

type S2TTWParams is (
// A64-VMSA exclusive parameters
    bit      ha,          // VTCR_EL2.HA
    bit      hd,          // VTCR_EL2.HD
    bit      sl2,         // V{S}TCR_EL2.SL2
    bit      ds,          // VTCR_EL2.DS
    bit      dl28,        // VTCR_EL2.D128
    bit      sw,          // VSTCR_EL2.SW
    bit      nsw,         // VTCR_EL2.NSW
    bit      sa,          // VSTCR_EL2.SA
    bit      nsa,         // VTCR_EL2.NSA
    bits(3)   ps,          // VTCR_EL2.PS
    bits(6)   txsz,        // V{S}TCR_EL2.T0SZ
    bit      fw,          // HCR_EL2.FWB
    bit      cmow,         // HCRX_EL2.CMOW
    bits(2)   skl,         // VTTBR_EL2.SKL
    bit      s2pie,        // VTCR_EL2.S2PIE
    S2PIRType s2pir,       // S2PIR_EL2
    bit      tl0,          // VTCR_EL2.TL0
    bit      tl1,          // VTCR_EL2.TL1
    bit      assuredonly, // VTCR_EL2.AssuredOnly
    bit      haft,         // VTCR_EL2.HAFT
    bit      emec,         // SCTLR2_EL2.EMEC
    bit      hdbss,        // VTCR_EL2.HDBSS

// A32-VMSA exclusive parameters
    bit      s,           // VTCR.S
    bits(4)   t0sz,        // VTCR.T0SZ

// Parameters common to both A64-VMSA & A32-VMSA if implemented (A64/A32)
    TGx      tgx,          // V{S}TCR_EL2.TG0 / Always TGx_4KB
    bits(2)   sl0,         // V{S}TCR_EL2.SL0 / VTCR.SL0
    bits(2)   irgn,        // VTCR_EL2.IRGN0 / VTCR.IRGN0
    bits(2)   orgn,        // VTCR_EL2.ORGNO / VTCR.ORGNO
    bits(2)   sh,          // VTCR_EL2.SH0 / VTCR.SH0
    bit      ee,           // SCTLR_EL2.EE / HSCTLR.EE
    bit      ptw,          // HCR_EL2.PTW / HCR.PTW
    bit      vm,           // HCR_EL2.VM / HCR.VM
)
```

## Library pseudocode for shared/translation/vmsa/SDFType

```
// SDFType
// =====
// Short-descriptor format type

enumeration SDFType {
    SDFType_Table,
    SDFType_Invalid,
    SDFType_Supersection,
    SDFType_Section,
    SDFType_LargePage,
    SDFType_SmallPage
};
```



## Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()
// =====
// Return the Security State of the given translation regime

SecurityState SecurityStateForRegime(Regime regime)
    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);
```

## 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

FullAddress StageOA(bits(64) ia, bit d128, TGx tgx, TTWState walkstate)
    // Output Address
    FullAddress oa;
    constant integer tsize = TranslationSize(d128, tgx, walkstate.level);
    constant integer csize = (if walkstate.contiguous == '1' then
                               ContiguousSize(d128, tgx, walkstate.level)
                               else 0);

    constant AddressSize ia_msb = tsize + csize;
    oa.paspace = walkstate.baseaddress.paspace;
    oa.address = walkstate.baseaddress.address<55:ia_msb>:ia<ia_msb-1:0>;

    return oa;
```

## Library pseudocode for shared/translation/vmsa/TGx

```
// TGx
// ===
// Translation granules sizes

enumeration TGx {
    TGx_4KB,
    TGx_16KB,
    TGx_64KB
};
```

## Library pseudocode for shared/translation/vmsa/TGxGranuleBits

```
// TGxGranuleBits()
// =====
// Retrieve the address size, in bits, of a granule

AddressSize TGxGranuleBits(TGx tgx)
    case tgx of
        when TGx\_4KB  return 12;
        when TGx\_16KB return 14;
        when TGx\_64KB return 16;
```

## Library pseudocode for shared/translation/vmsa/TLBContext

```
// TLBContext
// =====
// Translation context compared on TLB lookups and invalidations, promoting a TLB hit on match

type TLBContext is (
    SecurityState ss,
    Regime regime,
    bits(16) vmid,
    bits(16) asid,
    bit nG,
    PASpace ipaspace, // Used in stage 2 lookups & invalidations only
    boolean includes_s1,
    boolean includes_s2,
    boolean use_vmid,
    boolean includes_gpt,
    bits(64) ia, // Input Address
    TGx tg,
    bit cnp,
    integer level, // Assist TLBI level hints (FEAT_TTL)
    boolean isd128,
    bit xs // XS attribute (FEAT_XS)
)
```

## Library pseudocode for shared/translation/vmsa/TLBRecord

```
// TLBRecord
// =====
// Translation output as a TLB payload

type TLBRecord is (
    TLBContext context,
    TTWState walkstate,
    AddressSize blocksize, // Number of bits directly mapped from IA to OA
    integer contigsize, // Number of entries log 2 marking a contiguous output range
    bits(128) s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)
    bits(128) s2descriptor // 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 is (
    boolean istable,
    integer level,
    FullAddress baseaddress,
    bit contiguous,
    boolean s1assured, // Stage 1 Assured Translation Property
    bit s2assuredonly, // Stage 2 AssuredOnly attribute
    bit disch, // Stage 1 Disable Contiguous Hint
    bit nG,
    bit guardedpage,
    SDFTType sdftype, // AArch32 Short-descriptor format walk only
    bits(4) domain, // AArch32 Short-descriptor format walk only
    MemoryAttributes memattrs,
    Permissions permissions
)
```

## Library pseudocode for shared/translation/vmsa/TranslationRegime

```
// TranslationRegime()
// =====
// Select the translation regime given the target EL and PE state

Regime TranslationRegime(bits(2) el)
    if el == EL3 then
        return if ELUsingAArch32(EL3) then Regime\_EL30 else Regime\_EL3;
    elsif el == EL2 then
        return if ELIsInHost(EL2) then Regime\_EL20 else Regime\_EL2;
    elsif el == EL1 then
        return Regime\_EL10;
    elsif el == EL0 then
        if CurrentSecurityState() == SS\_Secure && ELUsingAArch32(EL3) then
            return Regime\_EL30;
        elsif ELIsInHost(EL0) then
            return Regime\_EL20;
        else
            return Regime\_EL10;
    else
        Unreachable();
```

## Library pseudocode for shared/translation/vmsa/TranslationSize

```
// TranslationSize()
// =====
// Compute the number of bits directly mapped from the input address
// to the output address

AddressSize TranslationSize(bit d128, TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    descsizelog2 = if d128 == '1' then 4 else 3;
    blockbits = (FINAL\_LEVEL - level) * (granulebits - descsizelog2);

    return granulebits + blockbits;
```

## Library pseudocode for shared/translation/vmsa/UseASID

```
// UseASID()
// =====
// Determine whether the translation context for the access requires ASID or is a global entry

boolean UseASID(TLBContext accesscontext)
    return HasUnprivileged(accesscontext.regime);
```

## Library pseudocode for shared/translation/vmsa/UseVMID

```
// UseVMID()
// =====
// Determine whether the translation context for the access requires VMID to match a TLB entry

boolean UseVMID(Regime regime)
    return regime == Regime\_EL10 && EL2Enabled();
```

## Library pseudocode for srmask/EffectiveACTLRMASK\_EL1

```
// EffectiveACTLRMASK_EL1()
// =====
// Return the effective value of ACTLRMASK_EL1.

ACTLR_EL1_Type EffectiveACTLRMASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEn == '0') then
        return Zeros(64);
    constant ACTLRMASK_EL1_Type mask = bits(64) IMPLEMENTATION_DEFINED "ACTLRMASK_EL1 layout";

    return mask;
```

## Library pseudocode for srmask/EffectiveACTLRMASK\_EL2

```
// EffectiveACTLRMASK_EL2()
// =====
// Return the effective value of ACTLRMASK_EL2.

ACTLR_EL2_Type EffectiveACTLRMASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    constant ACTLRMASK_EL2_Type mask = bits(64) IMPLEMENTATION_DEFINED "ACTLRMASK_EL2 layout";

    return mask;
```

## Library pseudocode for srmask/EffectiveCPACRMASK\_EL1

```
// EffectiveCPACRMASK_EL1()
// =====
// Return the effective value of CPACRMASK_EL1.

CPACR_EL1_Type EffectiveCPACRMASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEn == '0') then
        return Zeros(64);
    CPACR_EL1_Type mask = Ones(64);
    constant CPACRMASK_EL1_Type mask_reg = CPACRMASK_EL1;

    mask.TCPAC = mask_reg.TCPAC;
    mask.TAM = mask_reg.TAM;
    mask.E0POE = mask_reg.E0POE;
    mask.TTA = mask_reg.TTA;
    mask.SMEN = SignExtend(mask_reg.SMEN, 2);
    mask.FPEN = SignExtend(mask_reg.FPEN, 2);
    mask.ZEN = SignExtend(mask_reg.ZEN, 2);
    mask<32+: 32> = Zeros(32);
    mask<26+: 2> = Zeros(2);
    mask<22+: 2> = Zeros(2);
    mask<18+: 2> = Zeros(2);
    mask<0+: 16> = Zeros(16);

    return mask;
```

## Library pseudocode for srmask/EffectiveCPTRMASK\_EL2

```
// EffectiveCPTRMASK_EL2()
// =====
// Return the effective value of CPTRMASK_EL2.

CPTR_EL2_Type EffectiveCPTRMASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    CPTR_EL2_Type mask = Ones(64);
    constant CPTRMASK_EL2_Type mask_reg = CPTRMASK_EL2;

    if ELIsInHost(EL2) then
        mask.TCPAC = mask_reg.TCPAC;
        mask.TAM = mask_reg.TAM;
        mask.EOPOE = mask_reg.EOPOE;
        mask.TTA = mask_reg.TTA;
        mask.SMEN = SignExtend(mask_reg.SMEN, 2);
        mask.FPEN = SignExtend(mask_reg.FPEN, 2);
        mask.ZEN = SignExtend(mask_reg.ZEN, 2);
        mask<32+: 32> = Zeros(32);
        mask<26+: 2> = Zeros(2);
        mask<22+: 2> = Zeros(2);
        mask<18+: 2> = Zeros(2);
        mask<0+: 16> = Zeros(16);
    else
        mask.TCPAC = mask_reg.TCPAC;
        mask.TAM = mask_reg.TAM;
        mask.TTA = mask_reg.TTA;
        mask.TSM = mask_reg.TSM;
        mask.TFP = mask_reg.TFP;
        mask.TZ = mask_reg.TZ;
        mask<32+: 32> = Zeros(32);
        mask<21+: 9> = Zeros(9);
        mask<14+: 6> = Zeros(6);
        mask<13+: 1> = '0';
        mask<11+: 1> = '0';
        mask<9+: 1> = '0';
        mask<0+: 8> = Zeros(8);
    return mask;
```

## Library pseudocode for srmask/EffectiveSCTLR2MASK\_EL1

```
// EffectiveSCTLR2MASK_EL1()
// =====
// Return the effective value of SCTLR2MASK_EL1.

SCTLR2_EL1_Type EffectiveSCTLR2MASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEn == '0') then
        return Zeros(64);
    SCTLR2_EL1_Type mask = Ones(64);
    constant SCTLR2MASK_EL1_Type mask_reg = SCTLR2MASK_EL1;

    mask.CPTM0 = mask_reg.CPTM0;
    mask.CPTM = mask_reg.CPTM;
    mask.CPTA0 = mask_reg.CPTA0;
    mask.CPTA = mask_reg.CPTA;
    mask.EnPACM0 = mask_reg.EnPACM0;
    mask.EnPACM = mask_reg.EnPACM;
    mask.EnIDCP128 = mask_reg.EnIDCP128;
    mask.EASE = mask_reg.EASE;
    mask.EnANERR = mask_reg.EnANERR;
    mask.EnADERR = mask_reg.EnADERR;
    mask.NMEA = mask_reg.NMEA;
    mask<13+: 51> = Zeros(51);
    mask<0+: 2> = Zeros(2);
    return mask;
```

## Library pseudocode for srmask/EffectiveSCTLR2MASK\_EL2

```
// EffectiveSCTLR2MASK_EL2()
// =====
// Return the effective value of SCTLR2MASK_EL2.

SCTLR2_EL2_Type EffectiveSCTLR2MASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    SCTLR2_EL2_Type mask = Ones(64);
    constant SCTLR2MASK_EL2_Type mask_reg = SCTLR2MASK_EL2;

    mask.CPTM0 = mask_reg.CPTM0;
    mask.CPTM = mask_reg.CPTM;
    mask.CPTA0 = mask_reg.CPTA0;
    mask.CPTA = mask_reg.CPTA;
    mask.EnPACM0 = mask_reg.EnPACM0;
    mask.EnPACM = mask_reg.EnPACM;
    mask.EnIDCP128 = mask_reg.EnIDCP128;
    mask.EASE = mask_reg.EASE;
    mask.EnANERR = mask_reg.EnANERR;
    mask.EnADERR = mask_reg.EnADERR;
    mask.NMEA = mask_reg.NMEA;
    mask.EMEC = mask_reg.EMEC;
    mask<13+: 51> = Zeros(51);
    mask<0+: 1> = '0';
    return mask;
```



```

// EffectiveSCTLRMASK_EL1()
// =====
// Return the effective value of SCTLRMASK_EL1.

SCTLR_EL1_Type EffectiveSCTLRMASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEN == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEN == '0') then
        return Zeros(64);
    SCTLR_EL1_Type mask = Ones(64);
    constant SCTLRMASK_EL1_Type mask_reg = SCTLRMASK_EL1;

    mask.TIDCP = mask_reg.TIDCP;
    mask.SPINTMASK = mask_reg.SPINTMASK;
    mask.NMI = mask_reg.NMI;
    mask.EnTP2 = mask_reg.EnTP2;
    mask.TCSO = mask_reg.TCSO;
    mask.TCSO0 = mask_reg.TCSO0;
    mask.EPAN = mask_reg.EPAN;
    mask.EnALS = mask_reg.EnALS;
    mask.EnAS0 = mask_reg.EnAS0;
    mask.EnASR = mask_reg.EnASR;
    mask.TME = mask_reg.TME;
    mask.TME0 = mask_reg.TME0;
    mask.TMT = mask_reg.TMT;
    mask.TMT0 = mask_reg.TMT0;
    mask.TWEDEL = SignExtend(mask_reg.TWEDEL, 4);
    mask.TWEDEn = mask_reg.TWEDEn;
    mask.DSSBS = mask_reg.DSSBS;
    mask.ATA = mask_reg.ATA;
    mask.ATA0 = mask_reg.ATA0;
    mask.TCF = SignExtend(mask_reg.TCF, 2);
    mask.TCF0 = SignExtend(mask_reg.TCF0, 2);
    mask.ITFSB = mask_reg.ITFSB;
    mask.BT1 = mask_reg.BT1;
    mask.BT0 = mask_reg.BT0;
    mask.EnFPM = mask_reg.EnFPM;
    mask.MSCEn = mask_reg.MSCEn;
    mask.CMOW = mask_reg.CMOW;
    mask.EnIA = mask_reg.EnIA;
    mask.EnIB = mask_reg.EnIB;
    mask.LSMAOE = mask_reg.LSMAOE;
    mask.nTLSMD = mask_reg.nTLSMD;
    mask.EnDA = mask_reg.EnDA;
    mask.UCI = mask_reg.UCI;
    mask.EE = mask_reg.EE;
    mask.EOE = mask_reg.EOE;
    mask.SPAN = mask_reg.SPAN;
    mask.EIS = mask_reg.EIS;
    mask.IESB = mask_reg.IESB;
    mask.TSCXT = mask_reg.TSCXT;
    mask.WXN = mask_reg.WXN;
    mask.nTWE = mask_reg.nTWE;
    mask.nTWI = mask_reg.nTWI;
    mask.UCT = mask_reg.UCT;
    mask.DZE = mask_reg.DZE;
    mask.EnDB = mask_reg.EnDB;
    mask.I = mask_reg.I;
    mask.EOS = mask_reg.EOS;
    mask.EnRCTX = mask_reg.EnRCTX;
    mask.UMA = mask_reg.UMA;
    mask.SED = mask_reg.SED;
    mask.ITD = mask_reg.ITD;
    mask.nAA = mask_reg.nAA;
    mask.CP15BEN = mask_reg.CP15BEN;
    mask.SA0 = mask_reg.SA0;
    mask.SA = mask_reg.SA;
    mask.C = mask_reg.C;
    mask.A = mask_reg.A;
    mask.M = mask_reg.M;

```



```
mask<17+: 1> = '0';  
return mask;
```



```

// EffectiveSCTLRMASK_EL2()
// =====
// Return the effective value of SCTLRMASK_EL2.

SCTLR_EL2_Type EffectiveSCTLRMASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEN == '0' then return Zeros(64);
    SCTLR_EL2_Type mask = Ones(64);
    constant SCTLRMASK_EL2_Type mask_reg = SCTLRMASK_EL2;

    mask.TIDCP = mask_reg.TIDCP;
    mask.SPINTMASK = mask_reg.SPINTMASK;
    mask.NMI = mask_reg.NMI;
    mask.EnTP2 = mask_reg.EnTP2;
    mask.TCSO = mask_reg.TCSO;
    mask.TCSO0 = mask_reg.TCSO0;
    mask.EPAN = mask_reg.EPAN;
    mask.EnALS = mask_reg.EnALS;
    mask.EnAS0 = mask_reg.EnAS0;
    mask.EnASR = mask_reg.EnASR;
    mask.TME = mask_reg.TME;
    mask.TME0 = mask_reg.TME0;
    mask.TMT = mask_reg.TMT;
    mask.TMT0 = mask_reg.TMT0;
    mask.TWEDEL = SignExtend(mask_reg.TWEDEL, 4);
    mask.TWEDEn = mask_reg.TWEDEn;
    mask.DSSBS = mask_reg.DSSBS;
    mask.ATA = mask_reg.ATA;
    mask.ATA0 = mask_reg.ATA0;
    mask.TCF = SignExtend(mask_reg.TCF, 2);
    mask.TCF0 = SignExtend(mask_reg.TCF0, 2);
    mask.ITFSB = mask_reg.ITFSB;
    mask.BT = mask_reg.BT;
    mask.BT0 = mask_reg.BT0;
    mask.EnFPM = mask_reg.EnFPM;
    mask.MSCEn = mask_reg.MSCEn;
    mask.CMOW = mask_reg.CMOW;
    mask.EnIA = mask_reg.EnIA;
    mask.EnIB = mask_reg.EnIB;
    mask.LSMAOE = mask_reg.LSMAOE;
    mask.nTLSMD = mask_reg.nTLSMD;
    mask.EnDA = mask_reg.EnDA;
    mask.UCI = mask_reg.UCI;
    mask.EE = mask_reg.EE;
    mask.EOE = mask_reg.EOE;
    mask.SPAN = mask_reg.SPAN;
    mask.EIS = mask_reg.EIS;
    mask.IESB = mask_reg.IESB;
    mask.TSCXT = mask_reg.TSCXT;
    mask.WXN = mask_reg.WXN;
    mask.nTWE = mask_reg.nTWE;
    mask.nTWI = mask_reg.nTWI;
    mask.UCT = mask_reg.UCT;
    mask.DZE = mask_reg.DZE;
    mask.EnDB = mask_reg.EnDB;
    mask.I = mask_reg.I;
    mask.EOS = mask_reg.EOS;
    mask.EnRCTX = mask_reg.EnRCTX;
    mask.SED = mask_reg.SED;
    mask.ITD = mask_reg.ITD;
    mask.nAA = mask_reg.nAA;
    mask.CP15BEN = mask_reg.CP15BEN;
    mask.SA0 = mask_reg.SA0;
    mask.SA = mask_reg.SA;
    mask.C = mask_reg.C;
    mask.A = mask_reg.A;
    mask.M = mask_reg.M;
    mask<17+: 1> = '0';
    mask<9+: 1> = '0';
    return mask;

```

## Library pseudocode for srmask/EffectiveTCR2MASK\_EL1

```
// EffectiveTCR2MASK_EL1()
// =====
// Return the effective value of TCR2MASK_EL1.

TCR2_EL1_Type EffectiveTCR2MASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEn == '0') then
        return Zeros(64);
    TCR2_EL1_Type mask = Ones(64);
    constant TCR2MASK_EL1_Type mask_reg = TCR2MASK_EL1;

    mask.FNGNA1 = mask_reg.FNGNA1;
    mask.FNGNA0 = mask_reg.FNGNA0;
    mask.FNG1 = mask_reg.FNG1;
    mask.FNG0 = mask_reg.FNG0;
    mask.A2 = mask_reg.A2;
    mask.DisCH1 = mask_reg.DisCH1;
    mask.DisCH0 = mask_reg.DisCH0;
    mask.HAFT = mask_reg.HAFT;
    mask.PTTWI = mask_reg.PTTWI;
    mask.D128 = mask_reg.D128;
    mask.AIE = mask_reg.AIE;
    mask.POE = mask_reg.POE;
    mask.E0POE = mask_reg.E0POE;
    mask.PIE = mask_reg.PIE;
    mask.PnCH = mask_reg.PnCH;
    mask<22+: 42> = Zeros(42);
    mask<19+: 1> = '0';
    mask<12+: 2> = Zeros(2);
    mask<6+: 4> = Zeros(4);
    return mask;
```

## Library pseudocode for srmask/EffectiveTCR2MASK\_EL2

```
// EffectiveTCR2MASK_EL2()
// =====
// Return the effective value of TCR2MASK_EL2.

TCR2_EL2_Type EffectiveTCR2MASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    TCR2_EL2_Type mask = Ones(64);
    constant TCR2MASK_EL2_Type mask_reg = TCR2MASK_EL2;

    if !ELIsInHost(EL2) then
        mask.AMEC0 = mask_reg.AMEC0;
        mask.HAFT = mask_reg.HAFT;
        mask.PTTWI = mask_reg.PTTWI;
        mask.AIE = mask_reg.AIE;
        mask.POE = mask_reg.POE;
        mask.PIE = mask_reg.PIE;
        mask.PnCH = mask_reg.PnCH;
        mask<13+: 51> = Zeros(51);
        mask<5+: 5> = Zeros(5);
        mask<2+: 1> = '0';
    else
        mask.FNG1 = mask_reg.FNG1;
        mask.FNG0 = mask_reg.FNG0;
        mask.A2 = mask_reg.A2;
        mask.DisCH1 = mask_reg.DisCH1;
        mask.DisCH0 = mask_reg.DisCH0;
        mask.AMEC1 = mask_reg.AMEC1;
        mask.AMEC0 = mask_reg.AMEC0;
        mask.HAFT = mask_reg.HAFT;
        mask.PTTWI = mask_reg.PTTWI;
        mask.D128 = mask_reg.D128;
        mask.AIE = mask_reg.AIE;
        mask.POE = mask_reg.POE;
        mask.EOPOE = mask_reg.EOPOE;
        mask.PIE = mask_reg.PIE;
        mask.PnCH = mask_reg.PnCH;
        mask<19+: 45> = Zeros(45);
        mask<6+: 4> = Zeros(4);
    return mask;
```

## Library pseudocode for srmask/EffectiveTCRMASK\_EL1

```
// EffectiveTCRMASK_EL1()
// =====
// Return the effective value of TCRMASK_EL1.

TCR_EL1_Type EffectiveTCRMASK_EL1()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKEn == '0' then return Zeros(64);
    if EL2Enabled() && (!IsHCRXEL2Enabled() || HCRX_EL2.SRMASKEn == '0') then
        return Zeros(64);
    TCR_EL1_Type mask = Ones(64);
    constant TCRMASK_EL1_Type mask_reg = TCRMASK_EL1;

    mask.MTX1 = mask_reg.MTX1;
    mask.MTX0 = mask_reg.MTX0;
    mask.DS = mask_reg.DS;
    mask.TCMA1 = mask_reg.TCMA1;
    mask.TCMA0 = mask_reg.TCMA0;
    mask.EOPD1 = mask_reg.EOPD1;
    mask.EOPD0 = mask_reg.EOPD0;
    mask.NFD1 = mask_reg.NFD1;
    mask.NFD0 = mask_reg.NFD0;
    mask.TBID1 = mask_reg.TBID1;
    mask.TBID0 = mask_reg.TBID0;
    mask.HWU162 = mask_reg.HWU162;
    mask.HWU161 = mask_reg.HWU161;
    mask.HWU160 = mask_reg.HWU160;
    mask.HWU159 = mask_reg.HWU159;
    mask.HWU062 = mask_reg.HWU062;
    mask.HWU061 = mask_reg.HWU061;
    mask.HWU060 = mask_reg.HWU060;
    mask.HWU059 = mask_reg.HWU059;
    mask.HPD1 = mask_reg.HPD1;
    mask.HPD0 = mask_reg.HPD0;
    mask.HD = mask_reg.HD;
    mask.HA = mask_reg.HA;
    mask.TBI1 = mask_reg.TBI1;
    mask.TBI0 = mask_reg.TBI0;
    mask.AS = mask_reg.AS;
    mask.IPS = SignExtend(mask_reg.IPS, 3);
    mask.TG1 = SignExtend(mask_reg.TG1, 2);
    mask.SH1 = SignExtend(mask_reg.SH1, 2);
    mask.ORG1 = SignExtend(mask_reg.ORG1, 2);
    mask.IRG1 = SignExtend(mask_reg.IRG1, 2);
    mask.EPD1 = mask_reg.EPD1;
    mask.A1 = mask_reg.A1;
    mask.T1SZ = SignExtend(mask_reg.T1SZ, 6);
    mask.TG0 = SignExtend(mask_reg.TG0, 2);
    mask.SH0 = SignExtend(mask_reg.SH0, 2);
    mask.ORG0 = SignExtend(mask_reg.ORG0, 2);
    mask.IRG0 = SignExtend(mask_reg.IRG0, 2);
    mask.EPD0 = mask_reg.EPD0;
    mask.T0SZ = SignExtend(mask_reg.T0SZ, 6);
    mask<62+: 2> = Zeros(2);
    mask<35+: 1> = '0';
    mask<6+: 1> = '0';
    return mask;
```



```

// EffectiveTCRMASK_EL2()
// =====
// Return the effective value of TCRMASK_EL2.

TCR_EL2_Type EffectiveTCRMASK_EL2()
    if !IsFeatureImplemented(FEAT_SRMASK) then return Zeros(64);
    if HaveEL(EL3) && SCR_EL3.SRMASKE == '0' then return Zeros(64);
    TCR_EL2_Type mask = Ones(64);
    constant TCRMASK_EL2_Type mask_reg = TCRMASK_EL2;

    if !ELIsInHost(EL2) then
        mask.MTX = mask_reg.MTX;
        mask.DS = mask_reg.DS;
        mask.TCMA = mask_reg.TCMA;
        mask.TBID = mask_reg.TBID;
        mask.HWU62 = mask_reg.HWU62;
        mask.HWU61 = mask_reg.HWU61;
        mask.HWU60 = mask_reg.HWU60;
        mask.HWU59 = mask_reg.HWU59;
        mask.HPD = mask_reg.HPD;
        mask.HD = mask_reg.HD;
        mask.HA = mask_reg.HA;
        mask.TBI = mask_reg.TBI;
        mask.PS = SignExtend(mask_reg.PS, 3);
        mask.TG0 = SignExtend(mask_reg.TG0, 2);
        mask.SH0 = SignExtend(mask_reg.SH0, 2);
        mask.ORGNO = SignExtend(mask_reg.ORGNO, 2);
        mask.IRGNO = SignExtend(mask_reg.IRGNO, 2);
        mask.TOSZ = SignExtend(mask_reg.TOSZ, 6);
        mask<34+: 30> = Zeros(30);
        mask<31+: 1> = '0';
        mask<23+: 1> = '0';
        mask<19+: 1> = '0';
        mask<6+: 2> = Zeros(2);
    else
        mask.MTX1 = mask_reg.MTX1;
        mask.MTX0 = mask_reg.MTX0;
        mask.DS = mask_reg.DS;
        mask.TCMA1 = mask_reg.TCMA1;
        mask.TCMA0 = mask_reg.TCMA0;
        mask.EOPD1 = mask_reg.EOPD1;
        mask.EOPD0 = mask_reg.EOPD0;
        mask.NFD1 = mask_reg.NFD1;
        mask.NFD0 = mask_reg.NFD0;
        mask.TBID1 = mask_reg.TBID1;
        mask.TBID0 = mask_reg.TBID0;
        mask.HWU162 = mask_reg.HWU162;
        mask.HWU161 = mask_reg.HWU161;
        mask.HWU160 = mask_reg.HWU160;
        mask.HWU159 = mask_reg.HWU159;
        mask.HWU062 = mask_reg.HWU062;
        mask.HWU061 = mask_reg.HWU061;
        mask.HWU060 = mask_reg.HWU060;
        mask.HWU059 = mask_reg.HWU059;
        mask.HPD1 = mask_reg.HPD1;
        mask.HPD0 = mask_reg.HPD0;
        mask.HD = mask_reg.HD;
        mask.HA = mask_reg.HA;
        mask.TBI1 = mask_reg.TBI1;
        mask.TBI0 = mask_reg.TBI0;
        mask.AS = mask_reg.AS;
        mask.IPS = SignExtend(mask_reg.IPS, 3);
        mask.TG1 = SignExtend(mask_reg.TG1, 2);
        mask.SH1 = SignExtend(mask_reg.SH1, 2);
        mask.ORGNO1 = SignExtend(mask_reg.ORGNO1, 2);
        mask.IRGNO1 = SignExtend(mask_reg.IRGNO1, 2);
        mask.EPD1 = mask_reg.EPD1;
        mask.A1 = mask_reg.A1;
        mask.T1SZ = SignExtend(mask_reg.T1SZ, 6);
        mask.TG0 = SignExtend(mask_reg.TG0, 2);

```



```

mask.SH0 = SignExtend(mask_reg.SH0, 2);
mask.ORGNO = SignExtend(mask_reg.ORGNO, 2);
mask.IRGNO = SignExtend(mask_reg.IRGNO, 2);
mask.EPD0 = mask_reg.EPD0;
mask.TOSZ = SignExtend(mask_reg.TOSZ, 6);
mask<62+: 2> = Zeros(2);
mask<35+: 1> = '0';
mask<6+: 1> = '0';
return mask;

```

Internal version only: isa v01\_32, pseudocode v2024-12\_rel ; Build timestamp: 2024-12-16T10:54

Copyright © 2010-2024 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.