

Arm[®] A32/T32 Instruction Set

for A-profile architecture

arm

Arm A32/T32 Instruction Set for A-profile architecture

Copyright © 2010-2022 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 (2022-03)**.

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 implementations infringe any third party patents.

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.

This document may include technical inaccuracies or typographical errors.

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.

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 Arm corporate logo and words marked with [™] or [®] are registered trademarks or trademarks of Arm Limited (or its affiliates) in the US and/or elsewhere. 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 © 2010-2022 Arm Limited (or its affiliates). All rights reserved.

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

(LES-PRE-20349 version 21.0)

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 all Armv9-A features (including FEAT_RME and FEAT_SME) and the Armv8-A features, except for the Optional 64-bit external interface to the Performance Monitors, is at **Beta quality**. Beta quality means that all major features of the specification are described, some details might be missing.

The information relating to the Optional 64-bit external interface to the Performance Monitors is at **Alpha quality**. Alpha quality means that most major features of the specification are described in the manual, some features and 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.

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.

[CLREX](#): Clear-Exclusive.

[CLZ](#): Count Leading Zeros.

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

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

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

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

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

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

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

[CRC32](#): CRC32.

[CRC32C](#): CRC32C.

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

[DBG](#): Debug hint.

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

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

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

[DMB](#): Data Memory Barrier.

[DSB](#): Data Synchronization Barrier.

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

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

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

[ERET](#): Exception Return.

[ESB](#): Error Synchronization Barrier.

[HLT](#): Halting Breakpoint.

[HVC](#): Hypervisor Call.

[ISB](#): Instruction Synchronization Barrier.

[IT](#): If-Then.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[LDREX](#): Load Register Exclusive.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[LDRT](#): Load Register Unprivileged.

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

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

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

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

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

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

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

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

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

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

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

[MLS](#): Multiply and Subtract.

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

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

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

[MOVT](#): Move Top.

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

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

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

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

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

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

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

[MUL, MULS](#): Multiply.

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

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

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

[NOP](#): No Operation.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[QADD](#): Saturating Add.

[QADD16](#): Saturating Add 16.

[QADD8](#): Saturating Add 8.

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

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

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

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

[QSUB](#): Saturating Subtract.

[QSUB16](#): Saturating Subtract 16.

[QSUB8](#): Saturating Subtract 8.

[RBIT](#): Reverse Bits.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[SADD16](#): Signed Add 16.

[SADD8](#): Signed Add 8.

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

[SB](#): Speculation Barrier.

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

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

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

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

[SDIV](#): Signed Divide.

[SEL](#): Select Bytes.

[SETEND](#): Set Endianness.

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

[SEV](#): Send Event.

[SEVL](#): Send Event Local.

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

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

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

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

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

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

[SMC](#): Secure Monitor Call.

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

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

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

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

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

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

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

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

[SRS](#), [SRSDA](#), [SRSD](#), [SR](#), [SR](#): Store Return State.

[SSAT](#): Signed Saturate.

[SSAT16](#): Signed Saturate 16.

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

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

[SSUB16](#): Signed Subtract 16.

[SSUB8](#): Signed Subtract 8.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[STREX](#): Store Register Exclusive.

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

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

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

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

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

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

[STRT](#): Store Register Unprivileged.

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

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

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

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

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

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

[SVC](#): Supervisor Call.

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

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

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

[SXTB](#): Signed Extend Byte.

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

[SXTH](#): Signed Extend Halfword.

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

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

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

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

[TSB CSYNC](#): 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

ADC (S == 0)

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

ADCS (S == 1)

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

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

ADC (S == 0)

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

ADCS (S == 1)

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

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

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: 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 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 *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();
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register)

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

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

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

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

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

A1

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

ADC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

ADC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

ADCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

ADCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2

<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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ADC, ADCS (register-shifted register)

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

A1

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

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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																															

A1

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								

T1

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	0	1	1	1	1	0	imm3				Rd				imm8											

T3

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> 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 [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

For encoding T1 and T3: is the general-purpose destination register, encoded in the "Rd" field.

<label>

For encoding A1: 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 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.

<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 [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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

ADD (S == 0 && Rn != 11x1)

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

ADDS (S == 1 && Rn != 1101)

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

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); 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			

T1

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

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

```
d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); 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							

T2

```
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)
d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); 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

ADD (S == 0)

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

ADDS (S == 1 && Rd != 1111)

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

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

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

T4

```
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)
```

```
if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "ADD (SP plus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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

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();
    (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();
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			styp	0	Rm								
cond								Rn																							

ADD, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

ADD, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

ADDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

ADDS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

```
if Rn == '1101' then SEE "ADD (SP plus register)";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(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	

T1

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

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

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

T2 (!(DN == 1 && Rdn == 101))

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

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

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE "ADD (SP plus register)";
d = UInt(DN:Rdn); n = d; m = UInt(Rm); setflags = FALSE; (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

ADD, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ADD, shift or rotate by value (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>}

ADDS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

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

ADDS, shift or rotate by value (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>}

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

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 general-purpose source and destination register, encoded in the "DN:Rdn" field. If the PC is used, the instruction is a branch to the address calculated by the operation. This is a simple branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*. The assembler language allows <Rdn> to be specified once or twice in the assembler syntax. When used inside an IT block, and <Rdn> and <Rm> are in the range R0 to R7, <Rdn> must be specified once so that encoding T2 is preferred to encoding T1. In all other cases there is no difference in behavior when <Rdn> is specified once or twice.
- <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 ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see *Pseudocode description of operations on the AArch32 general-purpose registers and the PC*.
 - For the ADDS variant, the instruction performs an exception return, that restores *PSTATE* from *SPSR_<current_mode>*. Arm deprecates use of this instruction.
- For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. When used inside an IT block, <Rd> must be specified. When used outside an IT block, <Rd> is optional, and:
- If omitted, this register is the same as <Rn>.
 - If present, encoding T1 is preferred to encoding T2.
- 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":

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.

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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcV) = AddWithCarry(R[n], shifted, '0');
    R[d] = result;
    if setflags then
        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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

ADD (S == 0)

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

ADDS (S == 1)

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

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

T1

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

```
d = UInt(Rd); setflags = FALSE; 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						

T2

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

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

T3

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  (result, nzcvc) = AddWithCarry(SP, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

ADD, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

ADD, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

ADDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

ADDS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

```
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');  
(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		

T1

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

```
d = UInt(DM:Rdm); m = UInt(DM:Rdm); setflags = FALSE;  
(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

T2

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

```
if Rm == '1101' then SEE "encoding T1";
d = 13; m = UInt(Rm); setflags = FALSE;
(shift_t, shift_n) = (SType_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							

ADD, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ADD, shift or rotate by value (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>}

ADDS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

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

ADDS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

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

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

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](#).

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 [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

<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 ADD variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

- For the ADDS variant, the instruction performs an exception return, that restores *PSTATE* from *SPSR_<current_mode>*.

For 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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcvc) = AddWithCarry(SP, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

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

A2

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

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

T1

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

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

T2

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

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

T3

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

T3

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

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

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

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

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = if add then (Align(PC,4) + imm32) else (Align(PC,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																															

AND (S == 0)

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

ANDS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

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

AND (S == 0)

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

ANDS (S == 1 && Rd != 1111)

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

```
if Rd == '1111' && S == '1' then SEE "TST (immediate)";  
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);  
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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: 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 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 [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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

AND, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

AND, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

ANDS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

ANDS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2

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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AND, ANDS (register-shifted register)

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

A1

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

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
  shift_n = UInt(R[s]<7:0>);
  (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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								stype																			

MOV, shift or rotate by value

`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															

T2

`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	1	0	0	1	1	1	1	(0)	imm3			Rd			imm2		1	0	Rm				
S																stype															

MOV, shift or rotate by value

`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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASR (register)

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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								styp																			

Not flag setting

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																

Arithmetic shift right

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			
styp																S															

Not flag setting

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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (immediate)

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

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

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

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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							

MOVS, shift or rotate by value

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															

T2

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			

MOVS, shift or rotate by value

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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

ASRS (register)

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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																			

Flag setting

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																

Arithmetic shift right

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															

Flag setting

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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

B

Branch causes a branch to a target address.

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

A1

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

A1

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

```
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															

T1

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

```
if cond == '1110' then SEE "UDF";
if cond == '1111' then SEE "SVC";
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										

T2

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

```
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																															

T3

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)

```
if cond<3:1> == '111' then SEE "Related encodings";
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										

T4

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

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

```
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); 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(PC + 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	1	1	1	1
cond																																

A1

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

```
d = UInt(Rd); msbit = UInt(msb); 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				

T1

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

```
d = UInt(Rd); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BFI

Bit Field Insert copies any number of low order bits from a register into the same number of adjacent bits at any position in the destination register.

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

A1

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

A1

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

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd); n = UInt(Rn); msbit = UInt(msb); 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																																			

T1

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

```
if Rn == '1111' then SEE "BFC";
d = UInt(Rd); n = UInt(Rn); msbit = UInt(msb); lsbit = UInt(imm3:imm2);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.

- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <lsb> For 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

BIC (S == 0)

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

BICS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

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

BIC (S == 0)

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

BICS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);  
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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: 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 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 [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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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
!=	1	1	1	1	0	0	0	1	1	1	0	S	Rn			Rd			imm5			stype	0	Rm							
cond																															

BIC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

BIC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

BICS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

BICS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTypE_LSL, 0);
```

T2

<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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

BIC, BICS (register-shifted register)

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
  shift_n = UInt(R[s]<7:0>);
  (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
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							

T1

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

```
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](#). An 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);
```


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																															

A1

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

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

A2

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

```
imm32 = SignExtend(imm24:H:'0', 32); 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										

T1

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

```
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10:imm11:'0', 32);  
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

T2

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

```
if H == '1' then UNDEFINED;  
I1 = NOT(J1 EOR S); I2 = NOT(J2 EOR S); imm32 = SignExtend(S:I1:I2:imm10H:imm10L:'00', 32);  
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 [Standard assembler syntax fields](#).
For encoding A2: see [Standard assembler syntax fields](#). <c> must 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 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 = PC - 4;
    else
        LR = PC<31:1> : '1';
    bits(32) targetAddress;
    if targetInstrSet == InstrSet_A32 then
        targetAddress = Align(PC,4) + imm32;
    else
        targetAddress = PC + imm32;
    SelectInstrSet(targetInstrSet);
    BranchWritePC(targetAddress, BranchType_DIRCALL);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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)	(1)	0	0	1	1	Rm		
cond																															

A1

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

```
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)

T1

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

```
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();
    target = R[m];
    bits(32) next_instr_addr;
    if CurrentInstrSet() == InstrSet_A32 then
        next_instr_addr = PC - 4;
        LR = next_instr_addr;
    else
        next_instr_addr = PC - 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)	(1)	0	0	0	1	Rm		
cond																															

A1

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

```
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)

T1

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

```
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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

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

```
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)

T1

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

```
m = UInt(Rm);
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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		

CBNZ (op == 1)

```
CBNZ{<q>} <Rn>, <label>
```

CBZ (op == 0)

```
CBZ{<q>} <Rn>, <label>
```

```
n = UInt(Rn); imm32 = ZeroExtend(i:imm5:'0', 32); 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(PC + imm32);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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)

A1

```
CLREX{<c>}{<q>}
```

```
// 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)

T1

```
CLREX{<c>}{<q>}
```

```
// 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\(\)\);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

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

```
d = UInt(Rd); 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			

T1

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

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

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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	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	1	1	Rn				(0)	(0)	(0)	(0)	imm12											
cond																															

A1

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

```
n = UInt(Rn); 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	0	0	1	Rn				0	imm3				1	1	1	1	imm8									

T1

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

```
n = UInt(Rn); 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();  
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Rotate right with extend (imm5 == 00000 && stype == 11)

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

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

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

```
n = UInt(Rn); m = UInt(Rm);  
(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		

T1

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

```
n = UInt(Rn); m = UInt(Rm);  
(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	0	0	1	Rn			(0)	imm3			1	1	1	1	imm2		stype	Rm					

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

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

Shift or rotate by value (!(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>}
```

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

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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

CMN (register-shifted register)

Compare Negative (register-shifted register) adds a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	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)	Rs				0	stype	1	Rm				
cond																															

A1

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

```
n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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																															

A1

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

```
n = UInt(Rn); 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							

T1

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

```
n = UInt(Rn); 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							

T2

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

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

```
n = UInt(Rn); 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();
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Rotate right with extend (imm5 == 00000 && stype == 11)

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

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

```
n = UInt(Rn); m = UInt(Rm);  
(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	

T1

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

```
n = UInt(Rn); m = UInt(Rm);  
(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	1	N	Rm				Rn		

T2

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

```
n = UInt(N:Rn); m = UInt(Rm);  
(shift_t, shift_n) = (SRTYPE_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							

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

```
CMP{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(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>
```

```
n = UInt(Rn); m = UInt(Rm);
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if n == 15 || m == 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](#).
- <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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	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)	Rs				0	stype	1	Rm							
cond																															

A1

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

```
n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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				

Change mode (imod == 00 && M == 1)

```
CPS{<q>} #<mode> // (Cannot be conditional)
```

Interrupt disable (imod == 11 && M == 0)

```
CPSID{<q>} <iflags> // (Cannot be conditional)
```

Interrupt disable and change mode (imod == 11 && M == 1)

```
CPSID{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

Interrupt enable (imod == 10 && M == 0)

```
CPSIE{<q>} <iflags> // (Cannot be conditional)
```

Interrupt enable and change mode (imod == 10 && M == 1)

```
CPSIE{<q>} <iflags> , #<mode> // (Cannot be conditional)
```

```
if mode != '00000' && M == '0' then UNPREDICTABLE;
if (imod<1> == '1' && A:I:F == '000') || (imod<1> == '0' && A:I:F != '000') then UNPREDICTABLE;
enable = (imod == '10'); disable = (imod == '11'); changemode = (M == '1');
affectA = (A == '1'); affectI = (I == '1'); 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 != '00000' && M == '0'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: `changemode = TRUE`.
- The instruction executes as described, and the value specified by `mode` is ignored. There are no additional side-effects.

If `imod<1> == '1' && A:I:F == '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '0'`.
- The instruction behaves as if A:I:F has an UNKNOWN nonzero value.

If `imod<1> == '0' && A:I:F != '000'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction behaves as if `imod<1> == '1'`.
- The instruction behaves as if `A:I:F == '000'`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	1	im	(0)	A	I	F

Interrupt disable (im == 1)

`CPSID{<q>} <iflags> // (Not permitted in IT block)`

Interrupt enable (im == 0)

`CPSIE{<q>} <iflags> // (Not permitted in IT block)`

```
if A:I:F == '000' then UNPREDICTABLE;
enable = (im == '0'); disable = (im == '1'); changemode = FALSE;
affectA = (A == '1'); affectI = (I == '1'); affectF = (F == '1');
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					

Change mode (imod == 00 && M == 1)

```
CPS{<q>} #<mode> // (Not permitted in IT block)
```

Interrupt disable (imod == 11 && M == 0)

```
CPSID.W <iflags> // (Not permitted in IT block)
```

Interrupt disable and change mode (imod == 11 && M == 1)

```
CPSID{<q>} <iflags>, #<mode> // (Not permitted in IT block)
```

Interrupt enable (imod == 10 && M == 0)

```
CPSIE.W <iflags> // (Not permitted in IT block)
```

Interrupt enable and change mode (imod == 10 && M == 1)

```
CPSIE{<q>} <iflags>, #<mode> // (Not permitted in IT block)
```

```
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;
enable = (imod == '10'); disable = (imod == '11'); changemode = (M == '1');
affectA = (A == '1'); affectI = (I == '1'); 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 [Standard assembler syntax fields](#).

- <iflags> Is a sequence of one or more of the following, specifying which interrupt mask bits are affected:
- a** Sets the A bit in the instruction, causing the specified effect on `PSTATE.A`, the SError interrupt mask bit.
 - i** Sets the I bit in the instruction, causing the specified effect on `PSTATE.I`, the IRQ interrupt mask bit.
 - f** Sets the F bit in the instruction, causing the specified effect on `PSTATE.F`, the FIQ interrupt mask bit.
- <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 != EL0 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(mode);
else
    EncodingSpecificOperations();
    if PSTATE.EL != EL0 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(mode);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	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															

CRC32B (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

CRC32H (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

CRC32W (sz == 10)

CRC32W{<q>} <Rd>, <Rn>, <Rm>

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
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

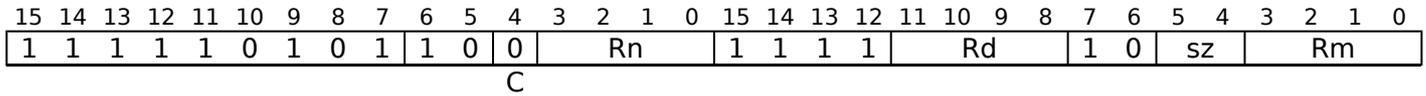
If `size == 64`, then one of the following behaviors must occur:

- 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



CRC32B (sz == 00)

CRC32B{<q>} <Rd>, <Rn>, <Rm>

CRC32H (sz == 01)

CRC32H{<q>} <Rd>, <Rn>, <Rm>

CRC32W (sz == 10)

CRC32W{<q>} <Rd>, <Rn>, <Rm>

```

if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
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 [Standard assembler syntax fields](#). An 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();

    acc = R[n];           // accumulator
    val = R[m]<size-1:0>; // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	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															

CRC32CB (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

CRC32CH (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

CRC32CW (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

```
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
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

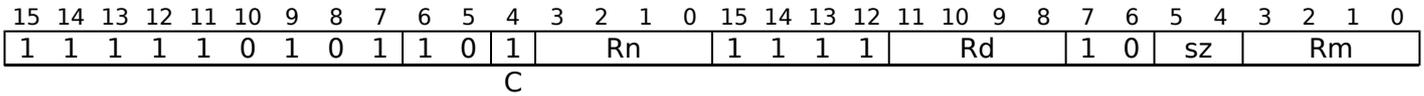
If `size == 64`, then one of the following behaviors must occur:

- 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



CRC32CB (sz == 00)

CRC32CB{<q>} <Rd>, <Rn>, <Rm>

CRC32CH (sz == 01)

CRC32CH{<q>} <Rd>, <Rn>, <Rm>

CRC32CW (sz == 10)

CRC32CW{<q>} <Rd>, <Rn>, <Rm>

```

if InITBlock() then UNPREDICTABLE;
if ! HaveCRCExt() then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
size = 8 << UInt(sz);
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 [Standard assembler syntax fields](#). An 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();

    acc = R[n]; // accumulator
    val = R[m]<size-1:0>; // input value
    poly = (if crc32c then 0x1EDC6F41 else 0x04C11DB7)<31:0>;
    tempacc = BitReverse(acc):Zeros(size);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 and data value prediction.

No instruction other than branch instructions and instructions that write to the PC appearing in program order after the CSDB can be speculatively executed using the results of any:

- Data value predictions of any instructions.
- PSTATE.{N,Z,C,V} predictions of any instructions other than conditional branch instructions and conditional instructions that write to the PC appearing in program order before the CSDB that have not been architecturally resolved.

Note

For purposes of the definition of CSDB, PSTATE.{N,Z,C,V} is not considered a data value. This definition permits:

- Control flow speculation before and after the CSDB.
- Speculative execution of conditional data processing instructions after the CSDB, unless they use the results of data value or PSTATE.{N,Z,C,V} predictions of instructions appearing in program order before the CSDB that have not been architecturally resolved.

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

A1

31	30	29	28	27	26	25	24	23	22	21	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	1	0	1	0	0	

cond

A1

CSDB{<c>}{<q>}

```
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

T1

CSDB{<c>}{<q>}

```
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	1	1	1	1	option				

cond

A1

```
DBG{<c>}{<q>} #<option>
```

```
// 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			

T1

```
DBG{<c>}{<q>} #<option>
```

```
// 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

T1

DCPS1

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

Operation

```
if !Halted() then UNDEFINED;

if EL2Enabled() && PSTATE.EL == EL0 then
    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 = SCTRLR.EE;
        if HavePANExt() && SCTRLR.SPAN == '0' then PSTATE.PAN = '1';
        LR_svc = bits(32) UNKNOWN;
        SPSR_svc = bits(32) UNKNOWN;
    else
        PSTATE.E = HSCTRLR.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 HavePANExt() && SCTRLR_EL1.SPAN == '0' then PSTATE.PAN = '1';
    if HaveUAOExt() 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;

    // SCTRLR_EL1.IESB might be ignored in Debug state.
    if HaveIESB() && SCTRLR_EL1.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        SynchronizeErrors();

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

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	0	1	0

T1

DCPS2

```
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 HavePANExt() && SCTLR_EL2.SPAN == '0' && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' then
        PSTATE.PAN = '1';
    if HaveUA0Ext() then PSTATE.UA0 = '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 HaveIESB() && SCTLR_EL2.IESB == '1' && !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        SynchronizeErrors();

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


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	0	1	1

T1

DCPS3

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

Operation

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

if ELUsingAArch32(EL3) then
    from_secure = CurrentSecurityState() == SS_Secure;
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if HavePANExt() 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;
    SP̄SR_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 HaveUAOExt() then PSTATE.UA0 = '0';

    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(64) UNKNOWN;
    SP̄SR_EL3 = bits(64) UNKNOWN;

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

    sync_errors = HaveIESB() && SCTLR_EL3.IESB == '1';
    if HaveDoubleFaultExt() && SCR_EL3.EA == '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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

```
DMB{<c>}{<q>} {<option>}
```

```
// 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			

T1

```
DMB{<c>}{<q>} {<option>}
```

```
// 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

DSB{<c>}{<q>} {<option>}

// 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			

T1

DSB{<c>}{<q>} {<option>}

// 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.

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 HaveFeatXS() && HaveFeatHCX() then
    nXS = (PSTATE.EL IN {EL0, EL1} && !ELUsingAArch32(EL2) &&
      IsHCRXEL2Enabled() && HCRX_EL2.FnXS == '1');
  else
    nXS = FALSE;
  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
    assert !(option IN {'0x00'});
    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;

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

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

EOR (S == 0)

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

EORS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

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

EOR (S == 0)

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

EORS (S == 1 && Rd != 1111)

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

```
if Rd == '1111' && S == '1' then SEE "TEQ (immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

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>	<p>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:</p> <ul style="list-style-type: none">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 <code>SPSR_<current_mode></code>. <p>For encoding T1: is the general-purpose destination register, encoded in the "Rd" field. If omitted, this register is the same as <Rn>.</p>
<Rn>	<p>For encoding A1: is the general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated.</p> <p>For encoding T1: is the general-purpose source register, encoded in the "Rn" field.</p>
<const>	<p>For encoding A1: an immediate value. See Modified immediate constants in A32 instructions for the range of values.</p> <p>For encoding T1: an immediate value. See Modified immediate constants in T32 instructions for the range of values.</p>

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

EOR, rotate right with extend (S == 0 && imm5 == 0000 && stype == 11)

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

EOR, shift or rotate by value (S == 0 && !(imm5 == 0000 && stype == 11))

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

EORS, rotate right with extend (S == 1 && imm5 == 0000 && stype == 11)

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

EORS, shift or rotate by value (S == 1 && !(imm5 == 0000 && stype == 11))

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

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

T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(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	0	S		Rn	(0)	imm3		Rd	imm2	stype		Rm									

EOR, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

EOR, shift or rotate by value (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>}

EORS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

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

EORS, shift or rotate by value (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>}

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

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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

EOR, EORS (register-shifted register)

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

A1

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

cond

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
  shift_n = UInt(R[s]<7:0>);
  (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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
!=	1	1	1	1	0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	0	(1)	(1)	(1)	(0)

cond

A1

ERET{<c>}{<q>}

// 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

T1

ERET{<c>}{<q>}

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
      new_pc_value = if PSTATE.EL == EL2 then ELR_hyp else R\[14\];
      AArch32.ExceptionReturn(new_pc_value, SPSR[]);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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. See Error Synchronization Barrier in the ARM(R) Reliability, Availability, and Serviceability (RAS) Specification, Armv8, for Armv8-A architecture profile.

If the RAS Extension 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	0	1	0	0	0	0	0	

cond

A1

ESB{<c>}{<q>}

```
if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
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

T1

ESB{<c>}{<q>}

```
if !HaveRASExt() then EndOfInstruction(); // Instruction executes as NOP
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.ESB0peration\(\);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) then AArch32.vESB0peration\(\);
    TakeUnmaskedSErrorInterrupts\(\);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	0	imm12										0	1	1	1	imm4					
cond																															

A1

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

```
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					

T1

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

```
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();  
boolean is_async = FALSE;  
Halt\(DebugHalt\_HaltInstruction, is\_async\);
```


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																															

A1

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

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

T1

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

```
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 [Standard assembler syntax fields](#). An HVC instruction must be unconditional.

<imm16> 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.

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.

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_GEN[].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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

ISB{<c>}{<q>} {<option>}

```
// 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			

T1

ISB{<c>}{<q>} {<option>}

```
// 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();
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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		

T1

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

```
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:

T

firstcond[0]

E

NOT firstcond[0]

<y> The condition for the third instruction in the IT block. If omitted and <x> is present, the "mask[2:0]" field is set to 0b100. If <y> is present it is encoded in the "mask[2]" field:

T

firstcond[0]

E

NOT firstcond[0]

<z> The condition for the fourth instruction in the IT block. If omitted and <y> is present, the "mask[1:0]" field is set to 0b10. If <z> is present, the "mask[0]" field is set to 1, and it is encoded in the "mask[1]" field:

T firstcond[0]

E NOT firstcond[0]

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

<cond> The condition for the first instruction in the IT block, encoded in the "firstcond" field. See [Condition codes](#) for the range of conditions available, and the encodings.

The conditions specified in an IT instruction must match those specified in the syntax of the instructions in its IT block. When assembling to A32 code, assemblers check IT instruction syntax for validity but do not generate assembled instructions for them. See [Conditional instructions](#).

Operation

```
EncodingSpecificOperations();  
AArch32.CheckITEnabled(mask);  
PSTATE.IT<7:0> = firstcond:mask;  
ShouldAdvanceIT = FALSE;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    R[t] = Mem0[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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    R[t] = ZeroExtend(Mem0[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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 4);
    R[t] = Mem0[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.

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
!=	1	1	1	1	0	0	0	1	1	1	0	1	Rn			Rt			(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)	
cond																															

A1

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

```
t = UInt(Rt); 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)		

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 1);
    R[t] = ZeroExtend(Mem0[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.

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																															

A1

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

```
t = UInt(Rt); t2 = t + 1; 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)

T1

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

```
t = UInt(Rt); t2 = UInt(Rt2); 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 [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. <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();
    address = R[n];
    AArch32.SetExclusiveMonitors(address, 8);
    value = Mem0[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(AccType\_ORDERED) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType\_ORDERED) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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
!	1	1	1	1	0	0	0	1	1	1	1	1	Rn			Rt			(1)	(1)	1	0	1	0	0	1	(1)	(1)	(1)	(1)	
cond																															

A1

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

```
t = UInt(Rt); 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)		

T1

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

```
t = UInt(Rt); 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();  
    address = R[n];  
    AArch32.SetExclusiveMonitors(address, 2);  
    R[t] = ZeroExtend(Mem0[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.

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

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    R[t] = ZeroExtend(Mem0[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 *Trapping general Non-secure System register accesses to debug registers*.

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																			

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

Unindexed (P == 0 && U == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

```
if Rn == '1111' then SEE "LDC (literal)";
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn); cp = 14;
imm32 = ZeroExtend(imm8:'00', 32); index = (P == '1'); add = (U == '1'); 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																															

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

Unindexed (P == 0 && U == 1 && W == 0)

```
LDC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

```
if Rn == '1111' then SEE "LDC (literal)";  
if P == '0' && U == '0' && W == '0' then UNDEFINED;  
n = UInt(Rn); cp = 14;  
imm32 = ZeroExtend(imm8:'00', 32); index = (P == '1'); add = (U == '1'); wback = (W == '1');
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rn> Is the general-purpose base register, encoded in the "Rn" field. If the PC is used, see [LDC \(literal\)](#).
- <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();  
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);  
    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 [Trapping general Non-secure System register accesses to debug registers](#).

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																															

A1 (!(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>

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1'); add = (U == '1'); cp = 14; 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							

T1 (!(P == 0 && U == 0 && W == 0))

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

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

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
index = (P == '1'); add = (U == '1'); cp = 14; 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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <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> 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> 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();
    offset_addr = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    address = if index then offset_addr else Align(PC,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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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.

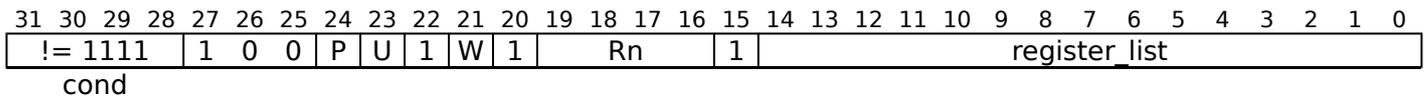
The registers loaded include the PC. The word loaded for the PC is treated as an address and a branch occurs to that address.

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



A1

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

```
n = UInt(Rn); registers = register_list;
wback = (W == '1'); increment = (U == '1'); 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:

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.

! 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;
    elsif PSTATE.M IN {M32\_User,M32\_System} then
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        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;
            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[]);

```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

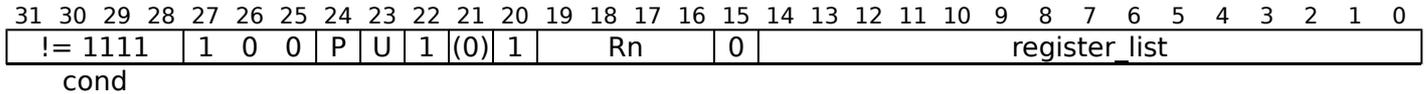
LDM (User registers)

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



A1

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

```
n = UInt(Rn); registers = register_list; increment = (U == '1'); 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
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDM, LDMIA, LDMFD

Load Multiple (Increment After, Full Descending) loads multiple registers from consecutive memory locations using an address from a base register. The consecutive memory locations start at this address, and the address just above the highest of those locations can optionally be written back to the base register.

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

Armv8.2 permits the deprecation of some Load Multiple ordering behaviors in AArch32 state, for more information see [FEAT LSMAOC](#). The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#). Related system instructions are [LDM \(User registers\)](#) and [LDM \(exception return\)](#).

This instruction is used by the alias [POP \(multiple registers\)](#).

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

A1

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

A1

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

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

```
n = UInt(Rn); registers = register_list; 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

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

T1

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

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

```
n = UInt(Rn); registers = '00000000':register_list; 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

T2

`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>

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

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

```
n = UInt(Rn); registers = P:M:register_list; 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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<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"> The LR must not be in the list. The instruction must be either outside any IT block, or the last instruction in an IT block.

Alias Conditions

Alias	Of variant	Is preferred when
POP (multiple registers)	T2	$W == '1' \ \&\& \ Rn == '1101' \ \&\& \ \text{BitCount}(P:M:register_list) > 1$
POP (multiple registers)	A1	$W == '1' \ \&\& \ Rn == '1101' \ \&\& \ \text{BitCount}(register_list) > 1$

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

A1

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

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

```
n = UInt(Rn); registers = register_list; 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 [Standard assembler syntax fields](#).

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

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list.

Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

A1

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

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

```
n = UInt(Rn); registers = register_list; 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												

T1

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

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

```
n = UInt(Rn); registers = P:M:register_list; 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<l3> == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction performs all of the loads using the specified addressing mode, but R13 is UNKNOWN.

If `P == '1' && M == '1'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction loads the register list and either R14 or R15, both R14 and R15, or neither of these registers.

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

Assembler Symbols

<code><c></code>	See Standard assembler syntax fields .
<code><q></code>	See Standard assembler syntax fields .
<code><Rn></code>	Is the general-purpose base register, encoded in the "Rn" field.
<code>!</code>	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<code><registers></code>	For 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: <ul style="list-style-type: none">• 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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

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

A1

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

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

```
n = UInt(Rn); registers = register_list; 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 [Standard assembler syntax fields](#).

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

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be loaded, separated by commas and surrounded by { and }. The PC can be in the list. Arm deprecates using these instructions with both the LR and the PC in the list.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																											

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDR (literal)";
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

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

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE; add = TRUE; 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							

T2

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

```
t = UInt(Rt); n = 13; imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE; add = TRUE; 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																															

T3

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

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

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); index = TRUE; add = TRUE;
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																															

Offset (P == 1 && U == 0 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDR (literal)";
if P == '1' && U == '1' && W == '0' then SEE "LDRT";
if P == '0' && W == '0' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn);
imm32 = ZeroExtend(imm8, 32); index = (P == '1'); add = (U == '1'); 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 [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 and T2: is the general-purpose register to be transferred, encoded in the "Rt" field.
 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 [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- <Rn> For encoding A1, T3 and T4: is the general-purpose base register, encoded in the "Rn" field. For PC use see [LDR \(literal\)](#).
 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":
- | 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
POP (single register)	A1 (post-indexed)	$P == '0' \ \&\& \ U == '1' \ \&\& \ W == '0' \ \&\& \ Rn == '1101' \ \&\& \ imm12 == '000000000100'$
POP (single register)	T4 (post-indexed)	$Rn == '1101' \ \&\& \ P == '0' \ \&\& \ U == '1' \ \&\& \ W == '1' \ \&\& \ imm8 == '00000100'$

Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about memory accesses see [Memory accesses](#).

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

A1

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

A1 (!!(P == 0 && W == 1))

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

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

```
if P == '0' && W == '1' then SEE "LDRT";  
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32);  
add = (U == '1'); 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					

T1

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

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

T2

```
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)
```

```
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); 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();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "LDRT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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	

T1

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
(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	1	!= 1111				Rt				0 0 0 0 0 0				imm2		Rm					
Rn																															

T2

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

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

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

+/- 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 CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if index then offset_addr else R[n];
    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();
      offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
      offset_addr = (R[n] + offset);
      address = offset_addr;
      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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																							

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDRB (literal)";
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

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

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5, 32);
index = TRUE; add = TRUE; 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	1	!= 1111				!= 1111				imm12											
Rn												Rt																			

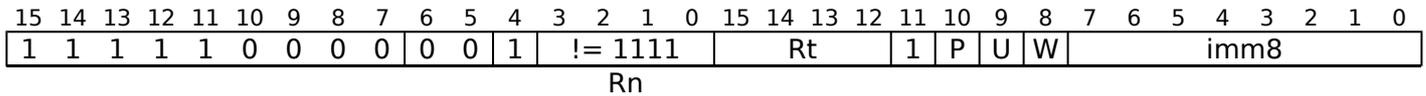
T2

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

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

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

T3



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
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;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); 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 [Standard assembler syntax fields](#).

<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 [LDRB \(literal\)](#).

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":

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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1 (!!(P == 0 && W == 1))

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

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

```
if P == '0' && W == '1' then SEE "LDRBT";  
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32);  
add = (U == '1'); 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																															

T1

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

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

```
if Rt == '1111' then SEE "PLD";  
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); 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();
    base = Align(PC,4);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "LDRBT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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	

T1

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(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				imm2	Rm						

Rn Rt

T2

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

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

```
if Rt == '1111' then SEE "PLD";
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if m == 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](#).

<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();
offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
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																															

A2

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	0	1	!= 1111				Rt				1	1	1	0	imm8							
Rn																															

T1

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

```
if Rn == '1111' then SEE "LDRB (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 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](#).

<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
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																				

Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]
```

Post-indexed (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

```
if Rn == '1111' then SEE "LDRD (literal)";
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); 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.

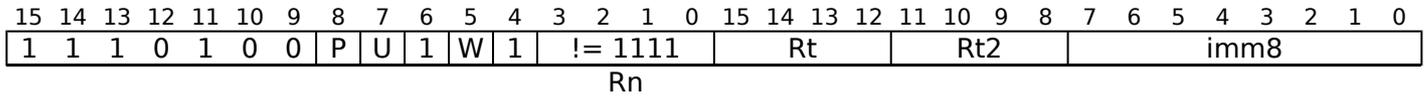
If `P == '0' && W == '1'`, then one of the following behaviors must occur:

- 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



Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #<+/-><imm>}]
```

Post-indexed (P == 0 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #<+/-><imm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #<+/-><imm>]!
```

```
if P == '0' && W == '0' then SEE "Related encodings";
if Rn == '1111' then SEE "LDRD (literal)";
t = UInt(Rt); t2 = UInt(Rt2); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1'); add = (U == '1'); wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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 [LDRD \(literal\)](#).

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm> 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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        data = MemA(address, 8);
        if BigEndian(AccType_ATOMIC) 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRD (literal)

Load Register Dual (literal) calculates an address from the PC value and an immediate offset, loads two words from memory, and writes them to two registers. For information about memory accesses see [Memory accesses](#).

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	(1)	U	1	(0)	0	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			
cond																															

A1

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)
```

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; imm32 = ZeroExtend(imm4H:imm4L, 32); 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							

T1 (!(P == 0 && W == 0))

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, <label> // (Normal form)
```

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [PC, #{+/-}<imm>] // (Alternative form)
```

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt); t2 = UInt(Rt2);
imm32 = ZeroExtend(imm8:'00', 32); add = (U == '1');
if t == 15 || t2 == 15 || t == t2 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [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.

<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();
  address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
  if address == Align(address, 8) then
    data = MemA[address,8];
    if BigEndian(AccType\_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
LDRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); 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();
  offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
  address = if index then offset_addr else R[n];
  if address == Align(address, 8) then
    data = MemA[address,8];
    if BigEndian(AccType\_ATOMIC) 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); 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							

T1

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

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
if t == 15 || n == 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](#).

<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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 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](#).
- <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();
    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.

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																															

A1

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

```
t = UInt(Rt); t2 = t + 1; 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)

T1

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

```
t = UInt(Rt); t2 = UInt(Rt2); 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 [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. <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();
    address = R[n];
    AArch32.SetExclusiveMonitors(address,8);
    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(AccType\_ATOMIC) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType\_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); n = UInt(Rn);
if t == 15 || n == 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](#).
- <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();
    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.

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																							

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDRH (literal)";
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

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

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE; add = TRUE; 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	1	!= 1111				!= 1111				imm12											
Rn												Rt																			

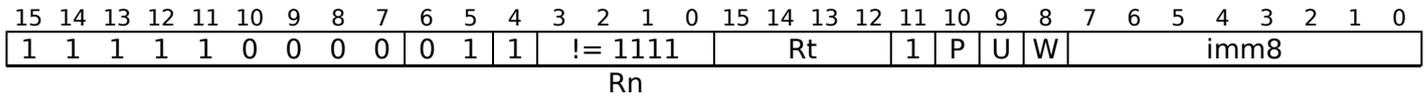
T2

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

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

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

T3



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
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;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); 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 [Standard assembler syntax fields](#).

<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 [LDRH \(literal\)](#).

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":

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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    data = MemU[address,2];
    if wback then R[n] = offset_addr;
    R[t] = ZeroExtend(data, 32);
else
  if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1 (!!(P == 0 && W == 1))

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

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

```
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt); imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1'); 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																															

T1

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

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

```
if Rt == '1111' then SEE "PLD (literal)";
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); 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();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "LDRHT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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		

T1

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(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	0	0	imm2		Rm	
Rn												Rt																			

T2

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

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

```
if Rn == 'l111' then SEE "LDRH (literal)";
if Rt == 'l111' then SEE "PLDW (register)";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if m == 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](#).

<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();
  offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  address = if index then offset_addr else R[n];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
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																															

A2

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE;
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

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

Rn

T1

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

```
if Rn == '1111' then SEE "LDRH (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 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](#).

<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
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																							

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDRSB (literal)";
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (W == '1'); 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

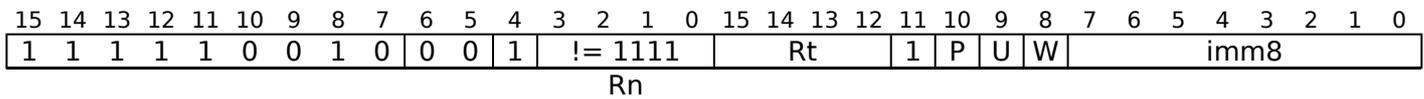
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1				1				1				0				0				1				!= 1111				!= 1111				imm12							
Rn																Rt																							

T1

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

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

T2



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```

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;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); 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 [Standard assembler syntax fields](#).

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. For PC use see [LDRSB \(literal\)](#).

+/- Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

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();
  offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSB (literal)

Load Register Signed Byte (literal) calculates an address from the PC value and an immediate offset, loads a byte from memory, sign-extends it to form a 32-bit word, and writes it to a register. For information about memory accesses see [Memory accesses](#).

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	P	U	1	W	1	1	1	1	1	Rt				imm4H				1	1	0	1	imm4L			
cond																															

A1 (!(P == 0 && W == 1))

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

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

```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt); imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1'); 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																															

T1

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

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

```
if Rt == '1111' then SEE "PLI";
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); 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();
    base = Align(PC,4);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

LDRSB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!

```
if P == '0' && W == '1' then SEE "LDRSBT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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		

T1

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

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

Rn Rt

T2

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

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

```
if Rt == '1111' then SEE "PLI";
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if m == 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](#).

<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();
  offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
  offset_addr = if add then (R[n] + offset) else (R[n] - offset);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
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																															

A2

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE;
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	0	1	!= 1111				Rt				1	1	1	0	imm8							
Rn																															

T1

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

```
if Rn == '1111' then SEE "LDRSB (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 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](#).

<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
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																							

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if Rn == '1111' then SEE "LDRSH (literal)";
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); 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

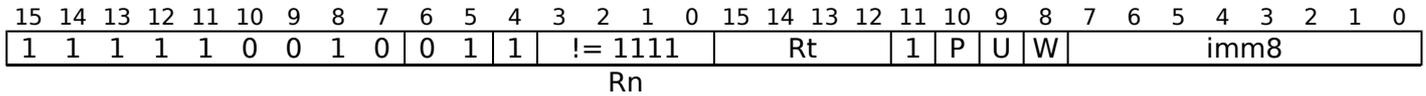
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
1				1				1				0				1				1				!= 1111				!= 1111				imm12							
Rn																Rt																							

T1

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

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

T2



Offset (Rt != 1111 && P == 1 && U == 0 && W == 0)

```
LDRSH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
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;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); 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();
  offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
  address = if index then offset_addr else R[n];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																																

A1 (!!(P == 0 && W == 1))

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

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

```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt); imm32 = ZeroExtend(imm4H:imm4L, 32);
add = (U == '1'); 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																															

T1

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

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

```
if Rt == '1111' then SEE "Related instructions";
t = UInt(Rt); imm32 = ZeroExtend(imm12, 32); 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();
    base = Align(PC,4);
    address = if add then (base + imm32) else (base - imm32);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LDRSH (register)

Load Register Signed Halfword (register) calculates an address from a base register value and an offset register value, loads a halfword from memory, sign-extends it to form a 32-bit word, and writes it to a register. The offset register value can be shifted left by 0, 1, 2, or 3 bits. For information about memory accesses see [Memory accesses](#).

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

A1

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

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "LDRSHT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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		

T1

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(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	0	imm2		Rm		
Rn												Rt																			

T2

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

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

```
if Rn == 'l111' then SEE "LDRSH (literal)";
if Rt == 'l111' then SEE "Related instructions";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if m == 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](#).

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.

+/- 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();
offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
offset_addr = if add then (R[n] + offset) else (R[n] - offset);
address = if index then offset_addr else R[n];
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
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																															

A2

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE;
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	0	1	1	!= 1111				Rt				1	1	1	0	imm8							
Rn																															

T1

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

```
if Rn == '1111' then SEE "LDRSH (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 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](#).

<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
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
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																															

A2

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

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
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

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

T1

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

```
if Rn == '1111' then SEE "LDR (literal)";
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 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](#).

<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
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    address = if postindex then R[n] else offset_addr;
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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															

MOV, shift or rotate by value

`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											

T2

`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															

MOV, shift or rotate by value

`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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSL (register)

Logical Shift Left (register) shifts a register value left by a variable number of bits, shifting in zeros, and writes the result to the destination register. The variable number of bits is read from the bottom byte of a register

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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								styp																			

Not flag setting

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																

Logical shift left

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	Rm				1	1	1	1	Rd				0	0	0	0	Rs				
styp																S															

Not flag setting

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 [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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (immediate)

Logical Shift Left, setting flags (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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

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

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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														

MOVS, shift or rotate by value

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												

T2

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	1	0	1	1	1	1	1	(0)	imm3			Rd			imm2		0	0	Rm				
S															imm5			stype													

MOVS, shift or rotate by value

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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSLS (register)

Logical Shift Left, setting flags (register) shifts a register value left by a variable number of bits, shifting in zeros, writes the result to the destination register, and updates the condition flags based on the result. The variable number of bits is read from the bottom byte of a register

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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																			

Flag setting

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																

Logical shift left

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	1	Rm				1	1	1	1	Rd				0	0	0	0	Rs			
styp																S															

Flag setting

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (immediate)

Logical Shift Right (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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								stype																			

MOV, shift or rotate by value

`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															

T2

`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	1	(0)	imm3			Rd			imm2		0	1	Rm				
S																stype															

MOV, shift or rotate by value

`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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSR (register)

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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																			

Not flag setting

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																

Logical shift right

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																

Not flag setting

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (immediate)

Logical Shift Right, setting flags (immediate) shifts a register value right by an immediate number of bits, shifting in zeros, and writes the result to the destination register.

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

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

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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								stypc																			

MOVS, shift or rotate by value

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

is equivalent to

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

and is always the preferred disassembly.

T2

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

T2

LSRS{<q>} {<Rd>}, <Rm>, #<imm> // (Outside IT block)

is equivalent to

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

MOVS, shift or rotate by value

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 [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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

LSRS (register)

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register-shifted register\)](#).
- The description of [MOV, MOVS \(register-shifted register\)](#) gives the operational pseudocode 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								stype																			

Flag setting

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															

Logical shift right

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			
stype																S															

Flag setting

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCR

Move to System register from general-purpose register or execute a System instruction. This instruction copies the value of a general-purpose register to a System register, or executes a System instruction.

The System register and System instruction descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MCR pseudocode does not show these possible traps.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1	1	1	1	0	opc1		0	CRn			Rt		1	1	1	coproc<0>		opc2		1	CRm									
cond										coproc<3:1>																					

A1

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
t = UInt(Rt); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1		0	CRn			Rt		1	1	1	coproc<0>		opc2		1	CRm							
										coproc<3:1>																					

T1

```
MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
t = UInt(Rt); cp = if coproc<0> == '0' then 14 else 15;
if t == 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 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 to7, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MCRR

Move to System register from two general-purpose registers. This instruction copies the values of two general-purpose registers to a System register.

The System register descriptions identify valid encodings for this instruction. Other encodings are UNDEFINED. For more information see [About the AArch32 System register interface](#) and [General behavior of System registers](#).

In an implementation that includes EL2, MCRR accesses to System registers can be trapped to Hyp mode, meaning that an attempt to execute an MCRR instruction in a Non-secure mode other than Hyp mode, that would be permitted in the absence of the Hyp trap controls, generates a Hyp Trap exception. For more information, see [EL2 configurable instruction enables, disables, and traps](#).

Because of the range of possible traps to Hyp mode, the MCRR pseudocode does not show these possible traps. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	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	1	0	0	Rt2			Rt			1	1	1	coproc<0>			opc1			CRm			
cond												coproc<3:1>																			

A1

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

```
t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 then UNPREDICTABLE;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	1	0	0	Rt2			Rt			1	1	1	coproc<0>			opc1			CRm				
																coproc<3:1>															

T1

MCRR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>

```
t = UInt(Rt); t2 = UInt(Rt2); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

Flag setting (S == 1)

MLAS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

Not flag setting (S == 0)

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); 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		Rn				!= 1111				Rd				0 0 0 0				Rm					

Ra

T1

MLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "MUL";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); setflags = FALSE;
if d == 15 || n == 15 || m == 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 addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations();
    operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    addend   = SInt(R[a]); // addend   = UInt(R[a]) produces the same final results
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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				Rn				Ra				Rd				0 0 0 1				Rm			

T1

MLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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();
operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
addend = SInt(R[a]); // addend = UInt(R[a]) produces the same final results
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

MOV (S == 0)

```
MOV{<c>}{<q>} <Rd>, #<const>
```

MOVS (S == 1)

```
MOVS{<c>}{<q>} <Rd>, #<const>
```

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

A2

```
MOV{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can not be represented in A1)
```

```
MOVW{<c>}{<q>} <Rd>, #<imm16> // (<imm16> can be represented in A1)
```

```
d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm4:imm12, 32);  
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			

T1

MOV<c>{<q>} <Rd>, #<imm8> // (Inside IT block)

MOVS{<q>} <Rd>, #<imm8> // (Outside IT block)

```
d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); 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														

MOV (S == 0)

MOV<c>.W <Rd>, #<const> // (Inside IT block, and <Rd>, <const> can be represented in T1)

MOV{<c>}{<q>} <Rd>, #<const>

MOVS (S == 1)

MOVS.W <Rd>, #<const> // (Outside IT block, and <Rd>, <const> can be represented in T1)

MOVS{<c>}{<q>} <Rd>, #<const>

```
d = UInt(Rd); setflags = (S == '1'); (imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	0	1	0	0		imm4		0	imm3															

T3

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)

```
d = UInt(Rd); setflags = FALSE; imm32 = ZeroExtend(imm4:i:imm3:imm8, 32);
if d == 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> 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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MOV, MOVS (register)

Move (register) copies a value from a register to the destination register.

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

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

- The MOV variant of the instruction is a branch. In the T32 instruction set (encoding T1) this is a simple branch, and in the A32 instruction set it is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
- The MOVS variant of the instruction performs an exception return without the use of the stack. In this case:
 - The PE branches to the address written to the PC, and restores *PSTATE* from *SPSR_<current_mode>*.
 - The PE checks *SPSR_<current_mode>* for an illegal return event. See [Illegal return events from AArch32 state](#).
 - The instruction is UNDEFINED in Hyp mode.
 - The instruction is CONSTRAINED UNPREDICTABLE in User mode and System mode.

This instruction is used by the aliases [ASRS \(immediate\)](#), [ASR \(immediate\)](#), [LSLS \(immediate\)](#), [LSL \(immediate\)](#), [LSRS \(immediate\)](#), [LSR \(immediate\)](#), [RORS \(immediate\)](#), [ROR \(immediate\)](#), [RRXS](#), and [RRX](#).

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

A1

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

MOV, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
MOV{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOV, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

MOVS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
MOVS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MOVS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

T1

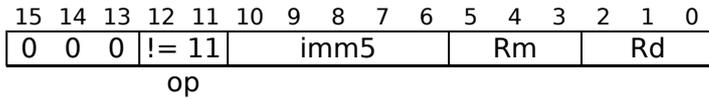
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

T1

```
MOV{<c>}{<q>} <Rd>, <Rm>
```

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

T2



T2

MOV<c>{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block)

MOVS{<q>} <Rd>, <Rm> {, <shift> #<amount>} // (Outside IT block)

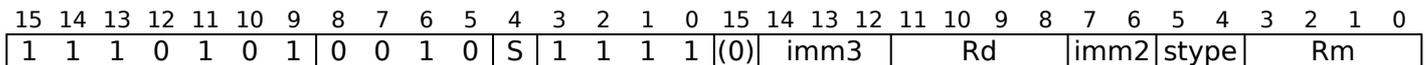
```
d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = DecodeImmShift(op, imm5);
if op == '00' && imm5 == '00000' && InITBlock() then UNPREDICTABLE;
```

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



MOV, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

MOV{<c>}{<q>} <Rd>, <Rm>, RRX

MOV, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

MOV{<c>}.W <Rd>, <Rm> {, LSL #0} // (<Rd>, <Rm> can be represented in T1)

MOV<c>{<q>}.W <Rd>, <Rm> {, <shift> #<amount>} // (Inside IT block, and <Rd>, <Rm>, <shift>, <amount> can be represented in T1)

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

MOVS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

MOVS{<c>}{<q>} <Rd>, <Rm>, RRX

MOVS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

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

```
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 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> 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
ASRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	<code>S == '1' && stype == '10'</code>
ASRS (immediate)	T2	<code>op == '10' && !InITBlock()</code>
ASR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	<code>S == '0' && stype == '10'</code>
ASR (immediate)	T2	<code>op == '10' && InITBlock()</code>

Alias	Of variant	Is preferred when
LSLS (immediate)	T3 (MOVS, shift or rotate by value)	<code>S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'</code>
LSLS (immediate)	A1 (MOVS, shift or rotate by value)	<code>S == '1' && imm5 != '00000' && stype == '00'</code>
LSLS (immediate)	T2	<code>op == '00' && imm5 != '00000' && !InITBlock()</code>
LSL (immediate)	T3 (MOV, shift or rotate by value)	<code>S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '00'</code>
LSL (immediate)	A1 (MOV, shift or rotate by value)	<code>S == '0' && imm5 != '00000' && stype == '00'</code>
LSL (immediate)	T2	<code>op == '00' && imm5 != '00000' && InITBlock()</code>
LSRS (immediate)	T3 (MOVS, shift or rotate by value), A1 (MOVS, shift or rotate by value)	<code>S == '1' && stype == '01'</code>
LSRS (immediate)	T2	<code>op == '01' && !InITBlock()</code>
LSR (immediate)	T3 (MOV, shift or rotate by value), A1 (MOV, shift or rotate by value)	<code>S == '0' && stype == '01'</code>
LSR (immediate)	T2	<code>op == '01' && InITBlock()</code>
RORS (immediate)	T3 (MOVS, shift or rotate by value)	<code>S == '1' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'</code>
RORS (immediate)	A1 (MOVS, shift or rotate by value)	<code>S == '1' && imm5 != '00000' && stype == '11'</code>
ROR (immediate)	T3 (MOV, shift or rotate by value)	<code>S == '0' && imm3:Rd:imm2 != '000xxxx00' && stype == '11'</code>
ROR (immediate)	A1 (MOV, shift or rotate by value)	<code>S == '0' && imm5 != '00000' && stype == '11'</code>
RRXS	T3 (MOVS, rotate right with extend)	<code>S == '1' && imm3 == '000' && imm2 == '00' && stype == '11'</code>
RRXS	A1 (MOVS, rotate right with extend)	<code>S == '1' && imm5 == '00000' && stype == '11'</code>
RRX	T3 (MOV, rotate right with extend)	<code>S == '0' && imm3 == '000' && imm2 == '00' && stype == '11'</code>
RRX	A1 (MOV, rotate right with extend)	<code>S == '0' && imm5 == '00000' && stype == '11'</code>

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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																	

Arithmetic shift right (op == 0100)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Inside IT block)
```

```
MOVS{<q>} <Rdm>, <Rdm>, ASR <Rs> // (Outside IT block)
```

Logical shift left (op == 0010)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Inside IT block)
```

```
MOVS{<q>} <Rdm>, <Rdm>, LSL <Rs> // (Outside IT block)
```

Logical shift right (op == 0011)

```
MOV<c>{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Inside IT block)
```

```
MOVS{<q>} <Rdm>, <Rdm>, LSR <Rs> // (Outside IT block)
```

Rotate right (op == 0111)

```
MOV<c>{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Inside IT block)
```

```
MOVS{<q>} <Rdm>, <Rdm>, ROR <Rs> // (Outside IT block)
```

```
if !(op IN {'0010', '0011', '0100', '0111'}) then SEE "Related encodings";
d = UInt(Rdm); m = UInt(Rdm); s = UInt(Rs);
setflags = !InITBlock(); 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							Rs

Flag setting (S == 1)

MOVS.W <Rd>, <Rm>, <shift> <Rs> // (Outside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in

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

Not flag setting (S == 0)

MOV<c>.W <Rd>, <Rm>, <shift> <Rs> // (Inside IT block, and <Rd>, <Rm>, <shift>, <Rs> can be represented in

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

```
d = UInt(Rd); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(stype);
if d == 15 || m == 15 || s == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

Related encodings: In encoding T1, for an op field value that is not described above, see [Data-processing \(two low registers\)](#).

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

Assembler Symbols

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

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

<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
ASRS (register)	A1 (flag setting)	S == '1' && stype == '10'
ASRS (register)	T1 (arithmetic shift right)	op == '0100' && !InITBlock()
ASRS (register)	T2 (flag setting)	stype == '10' && S == '1'
ASR (register)	A1 (not flag setting)	S == '0' && stype == '10'
ASR (register)	T1 (arithmetic shift right)	op == '0100' && InITBlock()
ASR (register)	T2 (not flag setting)	stype == '10' && S == '0'
LSLS (register)	A1 (flag setting)	S == '1' && stype == '00'
LSLS (register)	T1 (logical shift left)	op == '0010' && !InITBlock()
LSLS (register)	T2 (flag setting)	stype == '00' && S == '1'
LSL (register)	A1 (not flag setting)	S == '0' && stype == '00'

Alias	Of variant	Is preferred when
LSL (register)	T1 (logical shift left)	op == '0010' && InITBlock()
LSL (register)	T2 (not flag setting)	stype == '00' && S == '0'
LSRS (register)	A1 (flag setting)	S == '1' && stype == '01'
LSRS (register)	T1 (logical shift right)	op == '0011' && !InITBlock()
LSRS (register)	T2 (flag setting)	stype == '01' && S == '1'
LSR (register)	A1 (not flag setting)	S == '0' && stype == '01'
LSR (register)	T1 (logical shift right)	op == '0011' && InITBlock()
LSR (register)	T2 (not flag setting)	stype == '01' && S == '0'
RORS (register)	A1 (flag setting)	S == '1' && stype == '11'
RORS (register)	T1 (rotate right)	op == '0111' && !InITBlock()
RORS (register)	T2 (flag setting)	stype == '11' && S == '1'
ROR (register)	A1 (not flag setting)	S == '0' && stype == '11'
ROR (register)	T1 (rotate right)	op == '0111' && InITBlock()
ROR (register)	T2 (not flag setting)	stype == '11' && S == '0'

Operation

```

if ConditionPassed\(\) then
    EncodingSpecificOperations();
    shift_n = UInt(R[s]<7:0>);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

```
d = UInt(Rd); 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 1 0 0				imm4				0				imm3				Rd				imm8							

T1

```
MOVT{<c>}{<q>} <Rd>, #<imm16>
```

```
d = UInt(Rd); imm16 = imm4:i:imm3:imm8;
if d == 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.

<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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!=	1111	1	1	1	0	opc1		1	CRn		Rt		1	1	1	coproc<0>		opc2		1	CRm										
cond										coproc<3:1>																					

A1

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
t = UInt(Rt); cp = if coproc<0> == '0' then 14 else 15;
// Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1		1	CRn		Rt		1	1	1	coproc<0>		opc2		1	CRm								
										coproc<3:1>																					

T1

```
MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}
```

```
t = UInt(Rt); 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 to7, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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
!=	1	1	1	1	1	1	0	0	0	1	0	1	Rt2			Rt			1	1	1	coproc<0>			opc1			CRm			
cond											coproc<3:1>																				

A1

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

```
t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 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 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>																				

T1

```
MRRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <Rt2>, <CRm>
```

```
t = UInt(Rt); t2 = UInt(Rt2); cp = if coproc<0> == '0' then 14 else 15;
if t == 15 || t2 == 15 || t == t2 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 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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

MRS{<c>}{<q>} <Rd>, <spec_reg>

```
d = UInt(Rd); 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)	

T1

MRS{<c>}{<q>} <Rd>, <spec_reg>

```
d = UInt(Rd); read_spsr = (R == '1');
if d == 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.

<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[];
    else
        // CPSR has same bit assignments as SPSR, but with the IT, J, SS, IL, and T bits masked out.
        bits(32) mask = '11111000 11101111 00000011 11011111';
        psr_val = GetPSRFromPSTATE(AArch32\_NonDebugState) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	
cond																															

A1

MRS{<c>}{<q>} <Rd>, <banked_reg>

```
d = UInt(Rd); read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE;
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)	(0)	(0)	(0)	(0)	(0)	

T1

MRS{<c>}{<q>} <Rd>, <banked_reg>

```
d = UInt(Rd); read_spsr = (R == '1');
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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":

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

Operation

```
if ConditionPassed\(\) then
  EncodingSpecificOperations();
  if PSTATE.EL == EL0 then
    UNPREDICTABLE;
  else
    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
        integer m;
        BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
        case SYSm of
          when '00xxx' // Access the User mode registers
            m = UInt(SYSm<2:0>) + 8;
            R[d] = Rmode[m, M32\_User];
          when '01xxx' // Access the FIQ mode registers
            m = UInt(SYSm<2:0>) + 8;
            R[d] = Rmode[m, M32\_FIQ];
          when '1000x' // Access the IRQ mode registers
            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
            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
            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
            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\(\);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

MSR{<c>}{<q>} <banked_reg>, <Rn>

```
n = UInt(Rn); write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE;
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)						

T1

MSR{<c>}{<q>} <banked_reg>, <Rn>

```
n = UInt(Rn); write_spsr = (R == '1');
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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”:

R	M	M1	<banked_reg>
0	0	0000	R8_usr
0	0	0001	R9_usr
0	0	0010	R10_usr
0	0	0011	R11_usr
0	0	0100	R12_usr
0	0	0101	SP_usr
0	0	0110	LR_usr
0	0	0111	UNPREDICTABLE
0	0	1000	R8_fiq
0	0	1001	R9_fiq
0	0	1010	R10_fiq
0	0	1011	R11_fiq
0	0	1100	R12_fiq
0	0	1101	SP_fiq
0	0	1110	LR_fiq
0	0	1111	UNPREDICTABLE
0	1	0000	LR_irq
0	1	0001	SP_irq
0	1	0010	LR_svc
0	1	0011	SP_svc
0	1	0100	LR_abt
0	1	0101	SP_abt
0	1	0110	LR_und
0	1	0111	SP_und
0	1	10xx	UNPREDICTABLE
0	1	1100	LR_mon
0	1	1101	SP_mon
0	1	1110	ELR_hyp
0	1	1111	SP_hyp
1	0	0xxx	UNPREDICTABLE
1	0	10xx	UNPREDICTABLE
1	0	110x	UNPREDICTABLE
1	0	1110	SPSR_fiq
1	0	1111	UNPREDICTABLE
1	1	0000	SPSR_irq
1	1	0001	UNPREDICTABLE
1	1	0010	SPSR_svc
1	1	0011	UNPREDICTABLE
1	1	0100	SPSR_abt
1	1	0101	UNPREDICTABLE
1	1	0110	SPSR_und
1	1	0111	UNPREDICTABLE
1	1	10xx	UNPREDICTABLE
1	1	1100	SPSR_mon
1	1	1101	UNPREDICTABLE
1	1	1110	SPSR_hyp
1	1	1111	UNPREDICTABLE

<Rn> Is the general-purpose source register, encoded in the "Rn" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if PSTATE.EL == EL0 then
        UNPREDICTABLE;
    else
        mode = PSTATE.M;
        if write_spsr then
            SPSRaccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
            case SYSm of
                when '01110' SPSR_fiq = ZeroExtend(R[n]);
                when '10000' SPSR_irq = ZeroExtend(R[n]);
                when '10010' SPSR_svc = ZeroExtend(R[n]);
                when '10100' SPSR_abt = ZeroExtend(R[n]);
                when '10110' SPSR_und = ZeroExtend(R[n]);
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon = R[n];
                when '11110' SPSR_hyp = R[n];
            else
                integer m;
                BankedRegisterAccessValid(SYSm, mode); // Check for UNPREDICTABLE cases
                case SYSm of
                    when '00xxx' // Access the User mode registers
                        m = UInt(SYSm<2:0>) + 8;
                        Rmode[m,M32_User] = R[n];
                    when '01xxx' // Access the FIQ mode registers
                        m = UInt(SYSm<2:0>) + 8;
                        Rmode[m,M32_FIQ] = R[n];
                    when '1000x' // Access the IRQ mode registers
                        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
                        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
                        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
                        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();
                        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MSR (immediate)

Move immediate value to Special register moves selected bits of an immediate value to the corresponding bits in the [APSR](#), [CPSR](#), or [SPSR](#)_*<current_mode>*.

Because of the Do-Not-Modify nature of its reserved bits, the immediate form of MSR is normally only useful at the Application level for writing to [APSR_nzcvq](#) ([CPSR_f](#)).

If an MSR (immediate) moves selected bits of an immediate value to the [CPSR](#), the PE checks whether the value being written to [PSTATE.M](#) is legal. See [Illegal changes to PSTATE.M](#).

An MSR (immediate) executed in User mode:

- Is **CONSTRAINED UNPREDICTABLE** if it attempts to update the [SPSR](#).
- Otherwise, does not update any [CPSR](#) field that is accessible only at EL1 or higher,

An MSR (immediate) executed in System mode is **CONSTRAINED UNPREDICTABLE** if it attempts to update the [SPSR](#).

The [CPSR.E](#) bit is writable from any mode using an MSR instruction. Arm deprecates using this to change its value.

A1

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

A1 (!!(R == 0 && mask == 0000))

```
MSR{<c>}{<q>} <spec_reg>, #<imm>
```

```
if mask == '0000' && R == '0' then SEE "Related encodings";
imm32 = A32ExpandImm(imm12); 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 [nzcvq](#), [g](#), or [nzcvqg](#). These map to the following [CPSR](#)_*<fields>* values:

- [APSR_nzcvq](#) is the same as [CPSR_f](#) (mask== '1000').

- APSR_g is the same as CPSR_s (mask == '0100').
- APSR_nzcvqg is the same as CPSR_fs (mask == '1100').

Arm recommends the APSR_<bits> forms when only the N, Z, C, V, Q, and GE[3:0] bits are being written. For more information, see [The Application Program Status Register, APSR](#).

<imm> Is an immediate value. See [Modified immediate constants in A32 instructions](#) for the range of values.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    if write_spsr then
        if PSTATE.M IN {M32_User, M32_System} then
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

MSR{<c>}{<q>} <spec_reg>, <Rn>

```
n = UInt(Rn); 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)

T1

MSR{<c>}{<q>} <spec_reg>, <Rn>

```
n = UInt(Rn); write_spsr = (R == '1');
if mask == '0000' then UNPREDICTABLE;
if n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																																

Flag setting (S == 1)

```
MULS{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Not flag setting (S == 0)

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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	

T1

```
MUL<c>{<q>} <Rdm>, <Rn>{, <Rdm>} // (Inside IT block)
```

```
MULS{<q>} <Rdm>, <Rn>{, <Rdm>} // (Outside IT block)
```

```
d = UInt(Rdm); n = UInt(Rn); m = UInt(Rdm); 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			

T2

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

```
MUL{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = FALSE;  
if d == 15 || n == 15 || m == 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](#).
- <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();
    operand1 = SInt(R[n]); // operand1 = UInt(R[n]) produces the same final results
    operand2 = SInt(R[m]); // operand2 = UInt(R[m]) produces the same final results
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

MVN, MVNS (immediate)

Bitwise NOT (immediate) writes the bitwise inverse of an immediate value to the destination register.

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

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

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

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

A1

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

MVN (S == 0)

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
d = UInt(Rd); setflags = (S == '1');  
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

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

MVN (S == 0)

```
MVN{<c>}{<q>} <Rd>, #<const>
```

MVNS (S == 1)

```
MVNS{<c>}{<q>} <Rd>, #<const>
```

```
d = UInt(Rd); setflags = (S == '1');  
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);  
if d == 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> 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 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				styp	0	Rm					
cond																															

MVN, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
MVN{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVN, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

MVNS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
MVNS{<c>}{<q>} <Rd>, <Rm>, RRX
```

MVNS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

```
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(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			

T1

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

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

```
d = UInt(Rd); m = UInt(Rm); setflags = !InITBlock();
(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				

MVN, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

MVN{<c>}{<q>} <Rd>, <Rm>, RRX

MVN, shift or rotate by value (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>}

MVNS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

MVNS{<c>}{<q>} <Rd>, <Rm>, RRX

MVNS, shift or rotate by value (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>}

```
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 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> 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();
  (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

MVNS{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

Not flag setting (S == 0)

MVN{<c>}{<q>} <Rd>, <Rm>, <shift> <Rs>

```
d = UInt(Rd); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = NOT(shifted);
    R[d] = result;
    if setflags then
        PSTATE.N = result<31>;
        PSTATE.Z = IsZeroBit(result);
        PSTATE.C = carry;
    // PSTATE.V unchanged
```


NOP

No Operation does nothing. This instruction can be used for instruction alignment purposes.

Note

The timing effects of including a NOP instruction in a program are not guaranteed. It can increase execution time, leave it unchanged, or even reduce it. Therefore, NOP instructions are not suitable for timing loops.

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

A1

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

A1

NOP{<c>}{<q>}

```
// 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

T1

NOP{<c>}{<q>}

```
// 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

T2

NOP{<c>}.W

```
// 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.

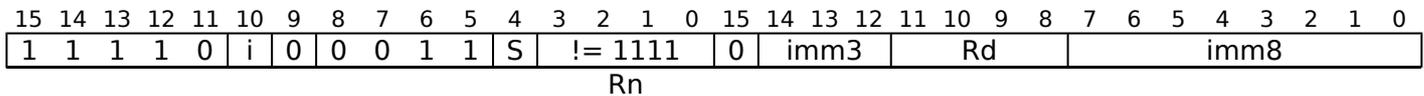
Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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



Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
if Rn == '1111' then SEE "MVN (immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if d == 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. 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();
    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

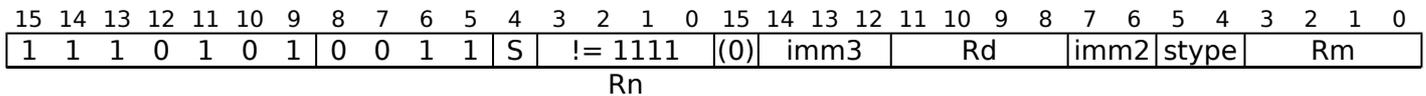
If CPSR.DIT is 1 and this instruction does not use R15 as either its source or destination:

- 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.

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



ORN, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ORN, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

ORNS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ORNS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

```
if Rn == '1111' then SEE "MVN (register)";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 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, 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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    result = R[n] OR NOT(shifted);
    R[d] = result;
    if setflags then
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

ORR (S == 0)

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

ORRS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

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

ORR (S == 0)

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

ORRS (S == 1)

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

```
if Rn == '1111' then SEE "MOV (immediate)";  
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1');  
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);  
if d == 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> 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 ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
 - For the ORRS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.
- For 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 [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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			styp	0	Rm						
cond																															

ORR, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

ORR, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

ORRS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

ORRS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

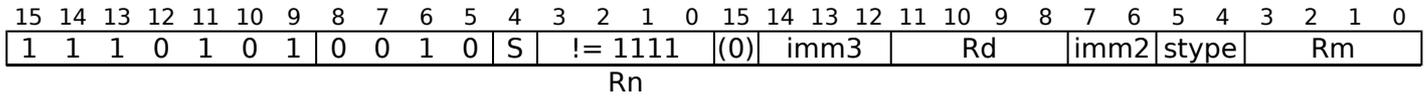
T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();  
(shift_t, shift_n) = (SRTypE_LSL, 0);
```

T2



ORR, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ORR, shift or rotate by value (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>}

ORRS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

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

ORRS, shift or rotate by value (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>}

```
if Rn == '1111' then SEE "Related encodings";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || m == 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](#).

Related encodings: [Data-processing \(shifted register\)](#)

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 ORR variant, the instruction is a branch to the address calculated by the operation. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).
 - For the ORRS variant, the instruction performs an exception return, that restores [PSTATE](#) from `SPSR_<current_mode>`.
 For 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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

PKHBT (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, LSL #<imm>}
```

PKHTB (tb == 1)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); tbform = (tb == '1');
(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																			

PKHBT (tb == 0)

```
PKHBT{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, LSL #<imm>} // (tbform == FALSE)
```

PKHTB (tb == 1)

```
PKHTB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ASR #<imm>} // (tbform == TRUE)
```

```
if S == '1' || T == '1' then UNDEFINED;
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); tbform = (tb == '1');
(shift_t, shift_n) = DecodeImmShift(tb:'0', imm3:imm2);
if d == 15 || n == 15 || m == 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, 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 0b00000.

1-31

Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted

Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32

Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

For 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 0b00000.

1-31

Left shift by specified number of bits, encoded as a binary number.

For PKHTB, it is one of:

omitted

Instruction is a pseudo-instruction and is assembled as though PKHBT{<c>}{<q>} <Rd>, <Rm>, <Rn> had been written.

1-32

Arithmetic right shift by specified number of bits. A shift by 32 bits is encoded as 0b00000. Other shift amounts are encoded as binary numbers.

Note

An assembler can permit <imm> = 0 to mean the same thing as omitting the shift, but this is not standard UAL and must not be used for disassembly.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

A1

PLD{<c>}{<q>} <label> // (Normal form)

PLD{<c>}{<q>} [PC, #<+/-><imm>] // (Alternative form)

```
imm32 = ZeroExtend(imm12, 32); 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							

T1

PLD{<c>}{<q>} <label> // (Preferred syntax)

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

```
imm32 = ZeroExtend(imm12, 32); 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();
    address = if add then (Align(PC,4) + imm32) else (Align(PC,4) - imm32);
    Hint\_PreloadData(address);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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

Preload read (R == 1)

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

Preload write (R == 0)

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

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = (U == '1'); 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

Preload read (W == 0)

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

Preload write (W == 1)

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

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); add = TRUE; is_pldw = (W == '1');
```

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	W	1	!= 1111	1	1	1	1	1	1	0	0	imm8										

Rn

Preload read (W == 0)

```
PLD{<c>}{<q>} [<Rn> {, #-<imm>}]
```

Preload write (W == 1)

```
PLDW{<c>}{<q>} [<Rn> {, #-<imm>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";  
n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); add = FALSE; 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 \(literal\)](#).
- +/- 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.

Operation

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

Preload read, optional shift or rotate (R == 1 && !(imm5 == 00000 && stype == 11))

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload read, rotate right with extend (R == 1 && imm5 == 00000 && stype == 11)

```
PLD{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

Preload write, optional shift or rotate (R == 0 && !(imm5 == 00000 && stype == 11))

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]
```

Preload write, rotate right with extend (R == 0 && imm5 == 00000 && stype == 11)

```
PLDW{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]
```

```
n = UInt(Rn); m = UInt(Rm); add = (U == '1'); is_pldw = (R == '0');
(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

Preload read (W == 0)

```
PLD{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

Preload write (W == 1)

```
PLDW{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]
```

```
if Rn == '1111' then SEE "PLD (literal)";
n = UInt(Rn); m = UInt(Rm); add = TRUE; is_pldw = (W == '1');
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if m == 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> 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.
- +/- 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();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    if is_pldw then
        Hint_PreloadDataForWrite(address);
    else
        Hint_PreloadData(address);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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											

A1

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

PLI{<c>}{<q>} <label> // (Normal form)

PLI{<c>}{<q>} [PC, #<+/-><imm>] // (Alternative form)

```
n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); 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																															

T1

PLI{<c>}{<q>} [<Rn> {, #<+><imm>}]

```
if Rn == '1111' then SEE "encoding T3";
```

```
n = UInt(Rn); imm32 = ZeroExtend(imm12, 32); 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																															

T2

PLI{<c>}{<q>} [<Rn> {, #<-><imm>}]

```
if Rn == '1111' then SEE "encoding T3";
```

```
n = UInt(Rn); imm32 = ZeroExtend(imm8, 32); 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	imm12											

T3

```
PLI{<c>}{<q>} <label> // (Preferred syntax)
```

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

```
n = 15; imm32 = ZeroExtend(imm12, 32); 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();
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    Hint_PreloadInstr(address);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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					

Rotate right with extend (imm5 == 00000 && stype == 11)

PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> , RRX]

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

PLI{<c>}{<q>} [<Rn>, {+/-}<Rm> {, <shift> #<amount>}]

```
n = UInt(Rn); m = UInt(Rm); add = (U == '1');
(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	0	0	imm2	Rm			

Rn

T1

PLI{<c>}{<q>} [<Rn>, {+}<Rm> {, LSL #<amount>}]

```
if Rn == '1111' then SEE "PLI (immediate, literal)";
n = UInt(Rn); m = UInt(Rm); add = TRUE;
(shift_t, shift_n) = (SRTType_LSL, UInt(imm2));
if m == 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> 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();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    address = if add then (R[n] + offset) else (R[n] - offset);
    Hint\_PreloadInstr(address);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

POP

Pop Multiple Registers from Stack loads multiple general-purpose registers from the stack, loading from consecutive memory locations starting at the address in SP, and updates SP to point just above the loaded data.

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

The registers loaded can include the PC, causing a branch to a loaded address. This is an interworking branch, see [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

T1

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

T1

```
POP{<c>}{<q>} <registers> // (Preferred syntax)
```

```
LDM{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
registers = P:'0000000':register_list; 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 = SP;
    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 SP = SP + 4*BitCount(registers);
    if registers<13> == '1' then SP = bits(32) UNKNOWN;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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																							

A1

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																							

T2

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 for this instruction.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	0	0	1	1	1	0	1	Rt				0	0	0	0	0	0	0	0	0	1	0	0
cond				P U				W				Rn				imm12															

Post-indexed

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								

Post-indexed

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

PSSBB

Physical Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same physical address.

The semantics of the Physical Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the PSSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
 - The store is to the same location as the load.
 - The store appears in program order before the PSSBB.
- When a load to a location appears in program order before the PSSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
 - The store is to the same location as the load.
 - The store appears in program order after the 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

A1

```
PSSBB{<q>}
```

```
// 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

T1

```
PSSBB{<q>}
```

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

Assembler Symbols

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

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    SpeculativeStoreBypassBarrierToPA();
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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							

T1

```
PUSH{<c>}{<q>} <registers> // (Preferred syntax)
```

```
STMDB{<c>}{<q>} SP!, <registers> // (Alternate syntax)
```

```
registers = '0':M:'000000':register_list; 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 = SP - 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();
    SP = SP - 4*BitCount(registers);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 for this instruction.

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

A1

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

A1

`PUSH{<c>}{<q>} <registers>`

is equivalent to

`STMDB{<c>}{<q>} SP!, <registers>`

and is the preferred disassembly when `BitCount(register_list) > 1`.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	1	0	0	1	0	1	1	0	1	(0)	M	register_list													
W							Rn				P																				

T1

`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](#), [STMED](#) gives the operational pseudocode for this instruction.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 for this instruction.

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

A1

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

Pre-indexed

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
																P				U	W	imm8									

Pre-indexed

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																															

A1

```
QADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>
```

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

```
QADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);  
if d == 15 || n == 15 || m == 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.
- <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																															

A1

QADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<15:0>) + SInt(R[m]<15:0>);
    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																															

A1

QADD8{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QADD8{<c>}{<q>} {<Rd>}, {<Rn>}, {<Rm>}

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    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);
```

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																															

A1

QASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

QDADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QDADD{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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.
- <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();
    (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																															

A1

QDSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QDSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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.
- <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();
    (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																															

A1

QSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    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																															

A1

QSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QSUB{<c>}{<q>} {<Rd>}, <Rm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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.
- <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) = SignedSat0(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																															

A1

QSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

QSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

QSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
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);
```

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																															

A1

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

```
d = UInt(Rd); 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			

T1

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

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

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
d = UInt(Rd); 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	

T1

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

```
d = UInt(Rd); 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	0	1	Rn			1	1	1	1	Rd			1	0	0	0	Rm				

T2

REV{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

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

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

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 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

REV16{<c>}{<q>} <Rd>, <Rm>

```
d = UInt(Rd); 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	

T1

REV16{<c>}{<q>} <Rd>, <Rm>

```
d = UInt(Rd); 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			

T2

REV16{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

REV16{<c>}{<q>} <Rd>, <Rm>

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

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 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
d = UInt(Rd); 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	

T1

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

```
d = UInt(Rd); 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			

T2

REVSH{<c>}.W <Rd>, <Rm> // (<Rd>, <Rm> can be represented in T1)

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

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

CONSTRAINED UNPREDICTABLE behavior

If $m \neq n$, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes with the additional decode: $m = \text{UInt}(Rn)$;
- The instruction executes with the additional decode: $m = \text{UInt}(Rm)$;
- 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 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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)	

Decrement After (P == 0 && U == 0)

RFEDA{<c>}{<q>} <Rn>{!} // (Preferred syntax)

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

Decrement Before (P == 1 && U == 0)

RFEDB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

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

Increment After (P == 0 && U == 1)

RFE{IA}{<c>}{<q>} <Rn>{!} // (Preferred syntax)

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

Increment Before (P == 1 && U == 1)

RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

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

```
n = UInt(Rn);
wback = (W == '1'); increment = (U == '1'); 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)	

T1

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)

```
n = UInt(Rn); wback = (W == '1'); increment = FALSE; 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)	

T2

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)

```
n = UInt(Rn); wback = (W == '1'); increment = TRUE; 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, RFEFA, 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;
    new_pc_value = MemA[address,4];
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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															

MOV, shift or rotate by value

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															

MOV, shift or rotate by value (!(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 [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.

<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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 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								stypc																			

Not flag setting

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																

Rotate right

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			
stypc S																															

Not flag setting

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RORS (immediate)

Rotate Right, setting flags (immediate) provides the value of the contents of a register rotated by a constant value. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

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

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

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

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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														

MOVS, shift or rotate by value

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

is equivalent to

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

and is always the preferred disassembly.

T3

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

MOVS, shift or rotate by value (!(imm3 == 000 && imm2 == 00))

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

is equivalent to

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

and is always the preferred disassembly.

Assembler Symbols

<c> See *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.
- <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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 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																			

Flag setting

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																

Rotate right

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															

Flag setting

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 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

RRX

Rotate Right with Extend provides the value of the contents of a register shifted right by one place, with the Carry flag shifted into bit[31].

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

- The encodings in this description are named to match the encodings of [MOV, MOVS \(register\)](#).
- The description of [MOV, MOVS \(register\)](#) gives the operational pseudocode 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														

MOV, rotate right with extend

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											

MOV, rotate right with extend

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 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														

MOVS, rotate right with extend

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

MOVS, rotate right with extend

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

RSB (S == 0)

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

RSBS (S == 1)

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

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

T1

RSB<c>{<q>} {<Rd>, }<Rn>, #0 // (Inside IT block)

RSBS{<q>} {<Rd>, }<Rn>, #0 // (Outside IT block)

```
d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); 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								

RSB (S == 0)

```
RSB<c>.W {<Rd>}, <Rn>, #0 // (Inside IT block)
```

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

RSBS (S == 1)

```
RSBS.W {<Rd>}, <Rn>, #0 // (Outside IT block)
```

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);  
if d == 15 || n == 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> 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 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 [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();  
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			styp	0	Rm							
cond																															

RSB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

RSB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

RSBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

RSBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(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		styp	Rm					

RSB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

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

RSB, shift or rotate by value (S == 0 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

RSBS, rotate right with extend (S == 1 && imm3 == 000 && imm2 == 00 && stype == 11)

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

RSBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11))

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if d == 15 || n == 15 || m == 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> 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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

RSC (S == 0)

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

RSCS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); 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();
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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
!=	1	1	1	1	0	0	0	0	1	1	1	S	Rn			Rd			imm5			stype	0	Rm							
cond																															

RSC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

RSC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

RSCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

RSCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
SADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

```
SADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    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 >= 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SB

Speculation Barrier is a barrier that controls speculation.

The semantics of the Speculation Barrier are that the execution, until the barrier completes, of any instruction that appears later in the program order than the barrier:

- Cannot be performed speculatively to the extent that such speculation can be observed through side-channels as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception.

In particular, any instruction that appears later in the program order than the barrier cannot cause a speculative allocation into any caching structure where the allocation of that entry could be indicative of any data value present in memory or in the registers.

The SB instruction:

- Cannot be speculatively executed as a result of control flow speculation or data value speculation.
- Can be speculatively executed as a result of predicting that a potentially exception generating instruction has not generated an exception. The potentially exception generating instruction can complete once it is known not to be speculative, and all data values generated by instructions appearing in program order before the SB instruction have their predicted values confirmed.

When the prediction of the instruction stream is not informed by data taken from the register outputs of the speculative execution of instructions appearing in program order after an uncompleted SB instruction, the SB instruction has no effect on the use of prediction resources to predict the instruction stream that is being fetched.

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)

A1

SB{<q>}

// 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)

T1

SB{<q>}

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																															

SBC (S == 0)

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

SBCS (S == 1)

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

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

SBC (S == 0)

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

SBCS (S == 1)

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

```
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);  
if d == 15 || n == 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> 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 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 *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();
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register)

Subtract with Carry (register) subtracts an optionally-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register.

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

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

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

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

A1

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

SBC, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

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

SBC, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

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

SBCS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

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

SBCS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

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

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

T1

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

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

```
d = UInt(Rdn); n = UInt(Rdn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTypE_LSL, 0);
```

T2

<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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SBC, SBCS (register-shifted register)

Subtract with Carry (register-shifted register) subtracts a register-shifted register value and the value of NOT (Carry flag) from a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

A1

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

Flag setting (S == 1)

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

Not flag setting (S == 0)

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

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(lsb); widthminus1 = UInt(widthm1);
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				

T1

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

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.

- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <lsb> For 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SDIV

Signed Divide divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. The condition flags are not affected.

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

A1

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

A1

SDIV{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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															

T1

SDIV{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>

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

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction executes as described, and the register specified by Ra becomes UNKNOWN.

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

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.
- <Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Overflow

If the signed integer division $0x80000000 / 0xFFFFFFFF$ is performed, the pseudocode produces the intermediate integer result $+2^{31}$, that overflows the 32-bit signed integer range. No indication of this overflow case is produced, and the 32-bit result written to <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();
    integer result;
    if SInt(R[m]) == 0 then
        result = 0;
    else
        result = RoundTowardsZero(Real(SInt(R[n])) / Real(SInt(R[m])));
    R[d] = result<31:0>;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SEL{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SEL{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, 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)

A1

```
SETEND{<q>} <endian_specifier> // (Cannot be conditional)
```

```
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)

T1

```
SETEND{<q>} <endian_specifier> // (Not permitted in IT block)
```

```
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)

A1

```
SETPAN{<q>} #<imm> // (Cannot be conditional)
```

```
if !HavePANExt() then UNDEFINED;  
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)

T1

```
SETPAN{<q>} #<imm> // (Not permitted in IT block)
```

```
if InITBlock() then UNPREDICTABLE;  
if !HavePANExt() then UNDEFINED;  
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;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	1	0	0

cond

A1

```
SEV{<c>}{<q>}
```

```
// 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

T1

```
SEV{<c>}{<q>}
```

```
// 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

T2

```
SEV{<c>}.W
```

```
// 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();
```

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	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	0	1

cond

A1

SEVL{<c>}{<q>}

```
// 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

T1

SEVL{<c>}{<q>}

```
// 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

T2

SEVL{<c>}.W

```
// 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();
```

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																															

A1

SHADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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	Rn				1	1	1	1	Rd				0	0	1	0	Rm						

T1

SHADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SHADD8{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SHADD8{<c>}{<q>} {<Rd>}, {<Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = SInt(R[n]<7:0>) + SInt(R[m]<7:0>);
    sum2 = SInt(R[n]<15:8>) + SInt(R[m]<15:8>);
    sum3 = SInt(R[n]<23:16>) + SInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

SHASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SHASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = SInt(R[n]<15:0>) - SInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SHSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SHSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = SInt(R[n]<15:0>) + SInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SHSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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	Rn				1	1	1	1	Rd				0	0	1	0	Rm						

T1

SHSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SHSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SHSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [Traps to Hyp mode of Non-secure EL1 execution of SMC instructions](#).

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
!=	1	1	1	1	0	0	0	1	0	1	1	0	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	0	1	1	1	imm4			
cond																															

A1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// 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)

T1

```
SMC{<c>}{<q>} {#}<imm4>
```

```
// 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.CheckForSMCUnDefOrTrap\(\);

    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.
                    c = ConstrainUnpredictable\(Unpredictable\_SMD\);
                    assert c IN {Constraint\_NOP, Constraint\_UNDEF};
                    if c == Constraint\_NOP then EndOfInstruction\(\);
                    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMLABB (M == 0 && N == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLABT (M == 1 && N == 0)

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATB (M == 0 && N == 1)

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATT (M == 1 && N == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
n_high = (N == '1'); 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																															

SMLABB (N == 0 && M == 0)

SMLABB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLABT (N == 0 && M == 1)

SMLABT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATB (N == 1 && M == 0)

SMLATB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLATT (N == 1 && M == 1)

SMLATT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMULBB, SMULBT, SMULTB, SMULTT";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
n_high = (N == '1'); m_high = (M == '1');
if d == 15 || n == 15 || m == 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 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();
operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
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';
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAD, SMLADX

Signed Multiply Accumulate Dual performs two signed 16 x 16-bit multiplications. It adds the products to a 32-bit accumulate operand.

Optionally, the instruction can exchange the halfwords of the second operand before performing the arithmetic. This produces top x bottom and bottom x top multiplication.

This instruction sets *PSTATE.Q* to 1 if the accumulate operation overflows. Overflow cannot occur during the multiplications.

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

A1

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

SMLAD (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADX (M == 1)

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
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															

SMLAD (M == 0)

SMLAD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLADX (M == 1)

SMLADX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMUAD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
m_swap = (M == '1');
if d == 15 || n == 15 || m == 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, 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();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

```
SMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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				Rn				RdLo				RdHi				0 0 0 0				Rm			

T1

```
SMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLALBB, SMLALBT, SMLALTB, SMLALTT

Signed Multiply Accumulate Long (halfwords) multiplies two signed 16-bit values to produce a 32-bit value, and accumulates this with a 64-bit value. The multiply acts on two signed 16-bit quantities, taken from either the bottom or the top half of their respective source registers. The other halves of these source registers are ignored. The 32-bit product is sign-extended and accumulated with a 64-bit accumulate value.

Overflow is possible during this instruction, but only as a result of the 64-bit addition. This overflow is not detected if it occurs. Instead, the result wraps around modulo 2^{64} .

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

A1

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

SMLALBB (M == 0 && N == 0)

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALBT (M == 1 && N == 0)

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTB (M == 0 && N == 1)

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTT (M == 1 && N == 1)

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm);
n_high = (N == '1'); 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				

SMLALBB (N == 0 && M == 0)

SMLALBB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALBT (N == 0 && M == 1)

SMLALBT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTB (N == 1 && M == 0)

SMLALTB{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALTT (N == 1 && M == 1)

SMLALTT{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm);
n_high = (N == '1'); 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 [Standard assembler syntax fields](#).

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

<RdLo> Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

<RdHi> Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.

<Rn> For 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();
operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMLALD (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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				

SMLALD (M == 0)

SMLALD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLALDX (M == 1)

SMLALDX{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLAWB, SMLAWT

Signed Multiply Accumulate (word by halfword) performs a signed multiply accumulate operation. The multiply acts on a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are added to a 32-bit accumulate value and the result is written to the destination register. The bottom 16 bits of the 48-bit product are ignored.

If overflow occurs during the addition of the accumulate value, the instruction sets *PSTATE.Q* to 1. No overflow can occur during the multiplication.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	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																															

SMLAWB (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLAWT (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); 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																															

SMLAWB (M == 0)

SMLAWB{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLAWT (M == 1)

SMLAWT{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMULWB, SMULWT";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); m_high = (M == '1');
if d == 15 || n == 15 || m == 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 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();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

SMLSD (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLSDX (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); 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															

SMLSD (M == 0)

SMLSD{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMLSDX (M == 1)

SMLSDX{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMUSD";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); m_swap = (M == '1');
if d == 15 || n == 15 || m == 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, 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();
operand2 = if m_swap then ROR(R[m],16) else R[m];
product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMLS LD, SMLS LD X

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																															

SMLS LD (M == 0)

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLS LD X (M == 1)

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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				

SMLS LD (M == 0)

SMLS LD{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

SMLS LD X (M == 1)

SMLS LD X{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = (product1 - product2) + SInt(R[dHi]:R[dLo]);
    R[dHi] = result<63:32>;
    R[dLo] = result<31:0>;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

SMMLA (R == 0)

SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMMLAR (R == 1)

SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); 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															

SMMLA (R == 0)

SMMLA{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMMLAR (R == 1)

SMMLAR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "SMMUL";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); round = (R == '1');
if d == 15 || n == 15 || m == 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 addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMMLS (R == 0)

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMMLSR (R == 1)

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); 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			

SMMLS (R == 0)

SMMLS{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

SMMLSR (R == 1)

SMMLSR{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra); round = (R == '1');
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 addend, encoded in the "Ra" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMMUL (R == 0)

SMMUL{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMMULR (R == 1)

SMMULR{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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			

SMMUL (R == 0)

SMMUL{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMMULR (R == 1)

SMMULR{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); round = (R == '1');
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMUAD (M == 0)

```
SMUAD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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			

SMUAD (M == 0)

```
SMUAD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

SMUADX (M == 1)

```
SMUADX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
operand2 = if m_swap then ROR(R[m],16) else R[m];
product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMULBB (M == 0 && N == 0)

SMULBB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULBT (M == 1 && N == 0)

SMULBT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULTB (M == 0 && N == 1)

SMULTB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULTT (M == 1 && N == 1)

SMULTT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
n_high = (N == '1'); 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			

SMULBB (N == 0 && M == 0)

SMULBB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULBT (N == 0 && M == 1)

SMULBT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULTB (N == 1 && M == 0)

SMULTB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMULTT (N == 1 && M == 1)

SMULTT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
n_high = (N == '1'); m_high = (M == '1');
if d == 15 || n == 15 || m == 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 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();
    operand1 = if n_high then R[n]<31:16> else R[n]<15:0>;
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Flag setting (S == 1)

```
SMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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			

T1

```
SMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SMULWB, SMULWT

Signed Multiply (word by halfword) multiplies a signed 32-bit quantity and a signed 16-bit quantity. The signed 16-bit quantity is taken from either the bottom or the top half of its source register. The other half of the second source register is ignored. The top 32 bits of the 48-bit product are written to the destination register. The bottom 16 bits of the 48-bit product are ignored. No overflow is possible during this instruction.

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

A1

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

SMULWB (M == 0)

```
SMULWB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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			

SMULWB (M == 0)

```
SMULWB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

SMULWT (M == 1)

```
SMULWT{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_high = (M == '1');
if d == 15 || n == 15 || m == 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 in the bottom or top half (selected by <y>), encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_high then R[m]<31:16> else R[m]<15:0>;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SMUSD (M == 0)

SMUSD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMUSDX (M == 1)

SMUSDX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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			

SMUSD (M == 0)

SMUSD{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

SMUSDX (M == 1)

SMUSDX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); m_swap = (M == '1');
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    operand2 = if m_swap then ROR(R[m],16) else R[m];
    product1 = SInt(R[n]<15:0>) * SInt(operand2<15:0>);
    product2 = SInt(R[n]<31:16>) * SInt(operand2<31:16>);
    result = product1 - product2;
    R[d] = result<31:0>;
    // Signed overflow cannot occur
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [Table G1-5](#) 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				

Decrement After (P == 0 && U == 0)

SRSDA{<c>}{<q>} SP{!}, #<mode>

Decrement Before (P == 1 && U == 0)

SRSDB{<c>}{<q>} SP{!}, #<mode>

Increment After (P == 0 && U == 1)

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

Increment Before (P == 1 && U == 1)

SRSIB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = (U == '1'); 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				

T1

SRSDB{<c>}{<q>} SP{!}, #<mode>

wback = (W == '1'); increment = FALSE; 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				

T2

SRS{IA}{<c>}{<q>} SP{!}, #<mode>

```
wback = (W == '1'); increment = TRUE; 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 PE mode descriptions](#).
- 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;

    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[];
    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;

      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[];
      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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Arithmetic shift right (sh == 1)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

```
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
(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							

Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>
```

Logical shift left (sh == 0)

```
SSAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}
```

```
if sh == '1' && (imm3:imm2) == '00000' then SEE "SSAT16";
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 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.
- <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 T1: 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();
  operand = Shift(R[n], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  (result, sat) = SignedSatQ(SInt(operand), saturate_to);
  R[d] = SignExtend(result, 32);
  if sat then
    PSTATE.Q = '1';
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSAT16

Signed Saturate 16 saturates two signed 16-bit values to a selected signed range.

This instruction sets *PSTATE.Q* to 1 if the operation saturates.

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

A1

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

A1

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SSAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm)+1;
if d == 15 || n == 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.

<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();
    (result1, sat1) = SignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (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																															

A1

SSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    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 >= 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SSBB

Speculative Store Bypass Barrier is a memory barrier which prevents speculative loads from bypassing earlier stores to the same virtual address under certain conditions.

The semantics of the Speculative Store Bypass Barrier are:

- When a load to a location appears in program order after the SSBB, then the load does not speculatively read an entry earlier in the coherence order for that location than the entry generated by the latest store satisfying all of the following conditions:
 - The store is to the same location as the load.
 - The store uses the same virtual address as the load.
 - The store appears in program order before the SSBB.
- When a load to a location appears in program order before the SSBB, then the load does not speculatively read data from any store satisfying all of the following conditions:
 - The store is to the same location as the load.
 - The store uses the same virtual address as the load.
 - The store appears in program order after the 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

A1

```
SSBB{<q>}
```

```
// 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	

T1

```
SSBB{<q>}
```

```
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();
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

SSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<15:0>) - SInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

SSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

SSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = SInt(R[n]<7:0>) - SInt(R[m]<7:0>);
    diff2 = SInt(R[n]<15:8>) - SInt(R[m]<15:8>);
    diff3 = SInt(R[n]<23:16>) - SInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *Trapping general Non-secure System register accesses to debug registers*.

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	0	imm8								
cond																															

Offset (P == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>{, #<+/-><imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>], #<+/-><imm>
```

Pre-indexed (P == 1 && W == 1)

```
STC{<c>}{<q>} p14, c5, [<Rn>, #<+/-><imm>]!
```

Unindexed (P == 0 && U == 1 && W == 0)

```
STC{<c>}{<q>} p14, c5, [<Rn>], <option>
```

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

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

STC{<c>}{<q>} p14, c5, [<Rn>, #{+/-}<imm>]!

Unindexed (P == 0 && U == 1 && W == 0)

STC{<c>}{<q>} p14, c5, [<Rn>], <option>

```
if P == '0' && U == '0' && W == '0' then UNDEFINED;
n = UInt(Rn); cp = 14;
imm32 = ZeroExtend(imm8:'00', 32); index = (P == '1'); add = (U == '1'); 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 [Standard assembler syntax fields](#).

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

<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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];

    // System register read from DBGDTRRXint.
    MemA[address,4] = AArch32.SysRegRead(cp, ThisInstr());
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); 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)	1	0	1	0	(1)	(1)	(1)	(1)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    Mem0[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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    Mem0[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																															

A1

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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			

T1

STLEX{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:
- 0**
If the operation updates memory.
- 1**
If the operation fails to update memory.
- <Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,4) then
        Mem0[address, 4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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			

T1

STLEXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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 [Standard assembler syntax fields](#).

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

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        Mem0[address, 1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
d = UInt(Rd); t = UInt(Rt); t2 = t+1; 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			

T1

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

```
d = UInt(Rd); t = UInt(Rt); t2 = UInt(Rt2); 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 [Standard assembler syntax fields](#).

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

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rt> For 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();
    address = R\[n\];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType\_ORDERED) then R\[t\]:R\[t2\] else R\[t2\]:R\[t\];
    if AArch32.ExclusiveMonitorsPass(address, 8) then
        Mem0[address, 8] = value;
        R\[d\] = ZeroExtend('0');
    else
        R\[d\] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

A1

STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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			

T1

STLEXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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 [Standard assembler syntax fields](#).

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

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated
- <Rd> is not updated.

A non word-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        Mem0[address, 2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

```
t = UInt(Rt); 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)

T1

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

```
t = UInt(Rt); 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();
    address = R[n];
    Mem0[address, 2] = R[t]<15:0>;
```

Operational information

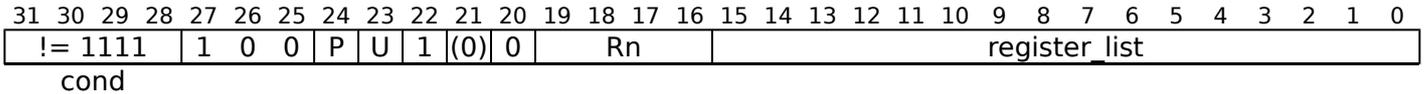
If CPSR.DIT is 1, the timing of this instruction is insensitive to the value of the data being loaded or stored.

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



A1

```
STM{<amode>}{<c>}{<q>} <Rn>, <registers>^
```

```
n = UInt(Rn); registers = register_list; increment = (U == '1'); 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;
    elsif PSTATE.M IN {M32_User,M32_System} then
        UNPREDICTABLE;
    else
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

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

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

```
n = UInt(Rn); registers = register_list; 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						

T1

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

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

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

CONSTRAINED UNPREDICTABLE behavior

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

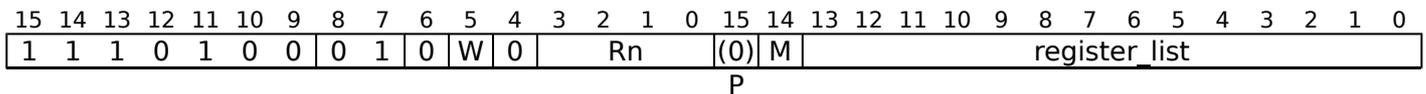
- The instruction is UNDEFINED.

- 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



T2

`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>

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

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

```
n = UInt(Rn); registers = P:M:register_list; 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 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>	<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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

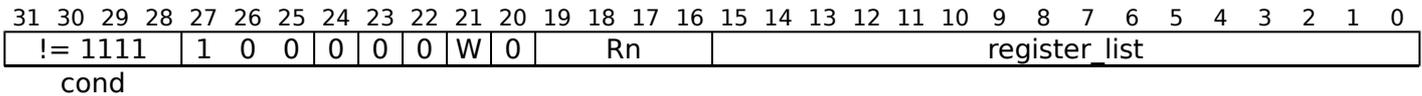
STMDA, STMED

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](#).

Arm v8.2 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



A1

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

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

```
n = UInt(Rn); registers = register_list; 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 [Standard assembler syntax fields](#).

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

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STMDB, STMFD

Store Multiple Decrement Before (Full Descending) stores multiple registers to consecutive memory locations using an address from a base register. The consecutive memory locations end just below this address, and the address of the first of those locations can optionally be written back to the base register.

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

Armv8.2 permits the deprecation of some Store Multiple ordering behaviors in AArch32 state, for more information see [FEAT LSMAOC](#). For details of related system instructions see [STM \(User registers\)](#).

This instruction is used by the alias [PUSH \(multiple registers\)](#).

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

A1

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

A1

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

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

```
n = UInt(Rn); registers = register_list; 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	0	0	1	0	0	W	0	Rn					(0)	M	register_list													
																	P															

T1

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

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

```
n = UInt(Rn); registers = P:M:register_list; 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

<code><c></code>	See Standard assembler syntax fields .
<code><q></code>	See Standard assembler syntax fields .
<code><Rn></code>	Is the general-purpose base register, encoded in the "Rn" field.
<code>!</code>	The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.
<code><registers></code>	For encoding A1: is a list of one or more registers to be stored, separated by commas and surrounded by { and }. The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list. If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register. For 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-R12, encoded in the "register_list" field, and can optionally contain the LR. If the LR is in the list, the "M" field is set to 1, otherwise it defaults to 0.

Alias Conditions

Alias	Of variant	Is preferred when
PUSH (multiple registers)	T1	<code>W == '1' && Rn == '1101' && BitCount(M:register_list) > 1</code>
PUSH (multiple registers)	A1	<code>W == '1' && Rn == '1101' && BitCount(register_list) > 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

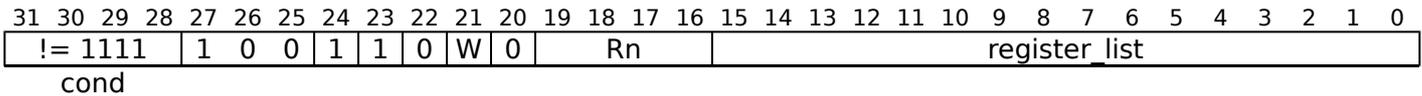
STMIB, STMFA

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



A1

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

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

```
n = UInt(Rn); registers = register_list; 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 [Standard assembler syntax fields](#).

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

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

! The address adjusted by the size of the data loaded is written back to the base register. If specified, it is encoded in the "W" field as 1, otherwise this field defaults to 0.

<registers> Is a list of one or more registers to be stored, separated by commas and surrounded by { and }.
The PC can be in the list. However, Arm deprecates the use of instructions that include the PC in the list.
If base register writeback is specified, and the base register is not the lowest-numbered register in the list, such an instruction stores an UNKNOWN value for the base register.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

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

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'00', 32);
index = TRUE; add = TRUE; 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

T2

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

```
t = UInt(Rt); n = 13; imm32 = ZeroExtend(imm8:'00', 32);
index = TRUE; add = TRUE; 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

T3

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

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

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; 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

Offset (P == 1 && U == 0 && W == 0)

```
STR{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '1' && U == '1' && W == '0' then SEE "STRT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); 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
PUSH (single register)	A1 (pre-indexed)	<code>P == '1' && U == '0' && W == '1' && Rn == '1101' && imm12 == '00000000100'</code>
PUSH (single register)	T4 (pre-indexed)	<code>Rn == '1101' && P == '1' && U == '0' && W == '1' && imm8 == '00000100'</code>

Operation

```
if CurrentInstrSet\(\) == InstrSet\_A32 then
  if ConditionPassed\(\) then
    EncodingSpecificOperations\(\);
    offset_addr = if add then (R\[n\] + imm32) else (R\[n\] - imm32);
    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\(\);
      offset_addr = if add then (R\[n\] + imm32) else (R\[n\] - imm32);
      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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STR (register)

Store Register (register) calculates an address from a base register value and an offset register value, stores a word from a register to memory. The offset register value can optionally be shifted. For information about memory accesses see [Memory accesses](#).

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

A1

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

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "STRT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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

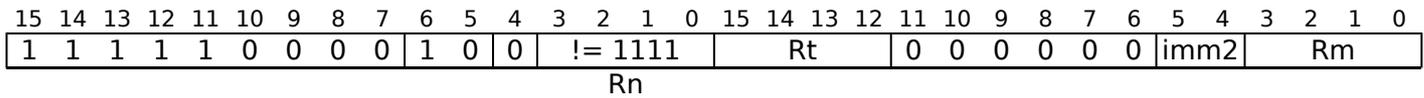
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm				Rn			Rt	

T1

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

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

T2



T2

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

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

```

if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

```

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 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":

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();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

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

Post-indexed (P == 0 && W == 0)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

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

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

T2

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

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

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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																															

Offset (P == 1 && U == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

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

Pre-indexed (P == 1 && W == 1)

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

```
if P == '1' && U == '1' && W == '0' then SEE "STRBT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

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

- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

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

Assembler Symbols

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

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

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For 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":

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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]
```

Post-indexed (P == 0 && W == 0)

```
STRB{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

Pre-indexed (P == 1 && W == 1)

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>{, <shift>}]!
```

```
if P == '0' && W == '1' then SEE "STRBT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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

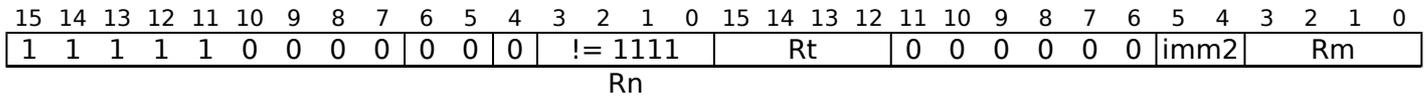
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	1	0	Rm				Rn			Rt	

T1

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]
```

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2



T2

```
STRB{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)
```

```
STRB{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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	+

+

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();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRBT

Store Register Byte Unprivileged stores a byte from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRBT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	1	1	0	Rn				Rt				imm12											
cond																															

A1

```
STRBT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}
```

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
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																															

A2

```
STRBT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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															
Rn																1	1	1	0	imm8											

T1

```
STRBT{<c>}{<q>} <Rt>, [<Rn> {, #+<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #+/-<imm>}]
```

Post-indexed (P == 0 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #+/-<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #+/-<imm>]!
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = '1'; 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																															

Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn> {, #{+/-}<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], #{+/-}<imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, #{+/-}<imm>]!
```

```
if P == '0' && W == '0' then SEE "Related encodings";
t = UInt(Rt); t2 = UInt(Rt2); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
index = (P == '1'); add = (U == '1'); wback = (W == '1');
if wback && (n == t || n == t2) then UNPREDICTABLE;
if n == 15 || t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15 || t2 == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && (n == t || n == t2)`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

If `wback && n == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes without writeback of the base address.
- The instruction uses the addressing mode described in the equivalent immediate offset instruction.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Related encodings: *Load/store dual, load/store exclusive, table branch*.

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rt> For 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":
- | 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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian(AccType_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]
```

Post-indexed (P == 0 && W == 0)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>], {+/-}<Rm>
```

Pre-indexed (P == 1 && W == 1)

```
STRD{<c>}{<q>} <Rt>, <Rt2>, [<Rn>, {+/-}<Rm>]!
```

```
if Rt<0> == '1' then UNPREDICTABLE;
t = UInt(Rt); t2 = t+1; n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); 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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rt> Is the first general-purpose register to be transferred, encoded in the "Rt" field. This register must be even-numbered and not R14.
- <Rt2> Is the second general-purpose register to be transferred. This register must be <R(t+1)>.
- <Rn> Is the general-purpose base register, encoded in the "Rn" field. The PC can be used in the offset variant, but this is deprecated.
- +/- Specifies the index register is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

- <Rm> Is the general-purpose index register, encoded in the "Rm" field.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations();
    offset_addr = if add then (R[n] + R[m]) else (R[n] - R[m]);
    address = if index then offset_addr else R[n];
    if address == Align(address, 8) then
        bits(64) data;
        if BigEndian(AccType_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 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

A1

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, {#}<imm>}]

```
d = UInt(Rd); t = UInt(Rt); n = UInt(Rn); 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							

T1

STREX{<c>}{<q>} <Rd>, <Rt>, [<Rn> {, #<imm>}]

```
d = UInt(Rd); t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rd>	Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is: 0 If the operation updates memory. 1 If the operation fails to update memory.
<Rt>	Is the general-purpose register to be transferred, encoded in the "Rt" field.
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.
<imm>	For 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();
    address = R[n] + imm32;
    if AArch32.ExclusiveMonitorsPass(address,4) then
        MemA[address,4] = R[t];
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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			

T1

STREXB{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

If AArch32.ExclusiveMonitorsPass() returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,1) then
        MemA[address,1] = R[t]<7:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); t2 = t+1; 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			

T1

```
STREXD{<c>}{<q>} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

```
d = UInt(Rd); t = UInt(Rt); t2 = UInt(Rt2); 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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rd> must not be the same as <Rn>, <Rt>, or <Rt2>.

<Rt> For 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 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();
    address = R[n];
    // Create doubleword to store such that R[t] will be stored at address and R[t2] at address+4.
    value = if BigEndian(AccType\_ATOMIC) then R[t]:R[t2] else R[t2]:R[t];
    if AArch32.ExclusiveMonitorsPass(address,8) then
        MemA[address,8] = value; R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); 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			

T1

STREXH{<c>}{<q>} <Rd>, <Rt>, [<Rn>]

```
d = UInt(Rd); t = UInt(Rt); n = UInt(Rn);
if d == 15 || t == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> Is the destination general-purpose register into which the status result of the store exclusive is written, encoded in the "Rd" field. The value returned is:

0

If the operation updates memory.

1

If the operation fails to update memory.

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

Aborts and alignment

If a synchronous Data Abort exception is generated by the execution of this instruction:

- Memory is not updated.
- <Rd> is not updated.

A non halfword-aligned memory address causes an Alignment fault Data Abort exception to be generated, subject to the following rules:

- If `AArch32.ExclusiveMonitorsPass()` returns TRUE, the exception is generated.
- Otherwise, it is IMPLEMENTATION DEFINED whether the exception is generated.

If `AArch32.ExclusiveMonitorsPass()` returns FALSE and the memory address, if accessed, would generate a synchronous Data Abort exception, it is IMPLEMENTATION DEFINED whether the exception is generated.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    address = R[n];
    if AArch32.ExclusiveMonitorsPass(address,2) then
        MemA[address,2] = R[t]<15:0>;
        R[d] = ZeroExtend('0');
    else
        R[d] = ZeroExtend('1');
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #<+/-><imm>}]
```

Post-indexed (P == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!
```

```
if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm4H:imm4L, 32);
index = (P == '1'); add = (U == '1'); 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			

T1

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

```
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm5:'0', 32);
index = TRUE; add = TRUE; 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																															

T2

```
STRH{<c>}.W <Rt>, [<Rn> {, #<+><imm>}] // (<Rt>, <Rn>, <imm> can be represented in T1)
```

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm12, 32);
index = TRUE; add = TRUE; wback = FALSE;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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																															

Offset (P == 1 && U == 0 && W == 0)

```
STRH{<c>}{<q>} <Rt>, [<Rn> {, #-<imm>}]
```

Post-indexed (P == 0 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>], #<+/-><imm>
```

Pre-indexed (P == 1 && W == 1)

```
STRH{<c>}{<q>} <Rt>, [<Rn>, #<+/-><imm>]!
```

```
if P == '1' && U == '1' && W == '0' then SEE "STRHT";
if Rn == '1111' || (P == '0' && W == '0') then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); imm32 = ZeroExtend(imm8, 32);
index = (P == '1'); add = (U == '1'); wback = (W == '1');
if t == 15 || (wback && n == t) then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If `t == 15`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.

- The store instruction performs the store using the specified addressing mode but the value corresponding to R15 is UNKNOWN.

If `wback && n == t`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The store instruction executes but the value stored is UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rt> Is the general-purpose register to be transferred, encoded in the "Rt" field.

<Rn> For 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":

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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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();
    offset_addr = if add then (R[n] + imm32) else (R[n] - imm32);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (P == 1 && W == 0)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]

Post-indexed (P == 0 && W == 0)

STRH{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>

Pre-indexed (P == 1 && W == 1)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+/-}<Rm>]!

```

if P == '0' && W == '1' then SEE "STRHT";
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = (P == '1'); add = (U == '1'); wback = (P == '0') || (W == '1');
(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

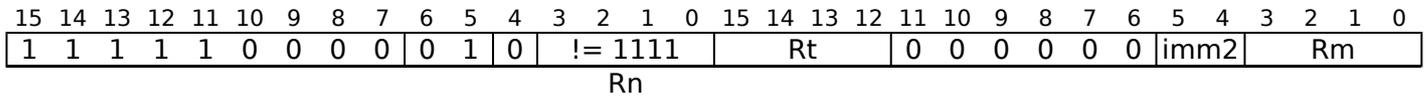
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0	1	0	1	0	0	1	Rm				Rn				Rt			

T1

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>]

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

T2



T2

STRH{<c>}.W <Rt>, [<Rn>, {+}<Rm>] // (<Rt>, <Rn>, <Rm> can be represented in T1)

STRH{<c>}{<q>} <Rt>, [<Rn>, {+}<Rm>{, LSL #<imm>}]

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm);
index = TRUE; add = TRUE; wback = FALSE;
(shift_t, shift_n) = (SRTYPE_LSL, UInt(imm2));
if t == 15 || m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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	+

+

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();
    offset = Shift(R[m], shift_t, shift_n, PSTATE.C);
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

STRHT{<c>}{<q>} <Rt>, [<Rn>] {, #{+/-}<imm>}

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm4H:imm4L, 32);
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																															

A2

```
STRHT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>
```

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE;
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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	0	0	0	1	0	!= 1111			Rt			1	1	1	0	imm8									
Rn																															

T1

```
STRHT{<c>}{<q>} <Rt>, [<Rn> {, #+<imm>}]
```

```
if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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
    offset = if register_form then R[m] else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

STRT

Store Register Unprivileged stores a word from a register to memory. For information about memory accesses see [Memory accesses](#).

The memory access is restricted as if the PE were running in User mode. This makes no difference if the PE is actually running in User mode.

STRT is UNPREDICTABLE in Hyp mode.

The T32 instruction uses an offset addressing mode, that calculates the address used for the memory access from a base register value and an immediate offset, and leaves the base register unchanged.

The A32 instruction uses a post-indexed addressing mode, that uses a base register value as the address for the memory access, and calculates a new address from a base register value and an offset and writes it back to the base register. The offset can be an immediate value or an optionally-shifted register value.

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	0	0	U	0	1	0	Rn				Rt				imm12											
cond																															

A1

```
STRT{<c>}{<q>} <Rt>, [<Rn>] {, #<+/-><imm>}
```

```
t = UInt(Rt); n = UInt(Rn); postindex = TRUE; add = (U == '1');
register_form = FALSE; imm32 = ZeroExtend(imm12, 32);
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																															

A2

```
STRT{<c>}{<q>} <Rt>, [<Rn>], {+/-}<Rm>{, <shift>}
```

```
t = UInt(Rt); n = UInt(Rn); m = UInt(Rm); postindex = TRUE; add = (U == '1');
register_form = TRUE; (shift_t, shift_n) = DecodeImmShift(stype, imm5);
if n == 15 || n == t || m == 15 then UNPREDICTABLE;
```

CONSTRAINED UNPREDICTABLE behavior

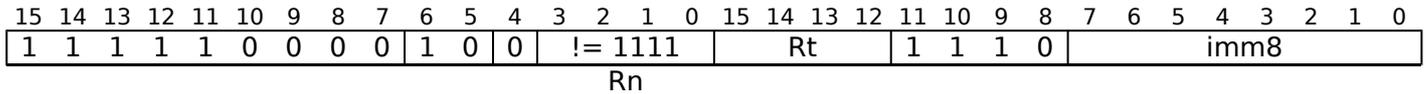
If `n == t`, then one of the following behaviors must occur:

- 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



T1

STRT{<c>}{<q>} <Rt>, [<Rn> {, #<+><imm>}]

```

if Rn == '1111' then UNDEFINED;
t = UInt(Rt); n = UInt(Rn); postindex = FALSE; add = TRUE;
register_form = FALSE; imm32 = ZeroExtend(imm8, 32);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13

```

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":

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*.
- +
- <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
    offset = if register_form then Shift(R[m], shift_t, shift_n, PSTATE.C) else imm32;
    offset_addr = if add then (R[n] + offset) else (R[n] - offset);
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SUB (immediate, from PC)

Subtract from PC subtracts an immediate value from the Align(PC, 4) value to form a PC-relative address, and writes the result to the destination register. Arm recommends that, where possible, software avoids using this alias

This is an alias of [ADR](#). This means:

- The encodings in this description are named to match the encodings of [ADR](#).
- The description of [ADR](#) gives the operational pseudocode 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																															

A2

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						

T2

SUB{<c>}{<q>} <Rd>, PC, #<imm12>

is equivalent to

[ADR](#){<c>}{<q>} <Rd>, <label>

and is the preferred disassembly when `i:imm3:imm8 == '000000000000'`.

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> For 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 [Pseudocode description of operations on the AArch32 general-purpose registers and the PC](#).

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 [Modified immediate constants in A32 instructions](#).

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

SUB (S == 0 && Rn != 11x1)

```
SUB{<c>}{<q>} {<Rd>}, <Rn>, #<const>
```

SUBS (S == 1 && Rn != 1101)

```
SUBS{<c>}{<q>} {<Rd>}, <Rn>, #<const>
```

```
if Rn == '1111' && S == '0' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); 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						

T1

```
SUB<c>{<q>} <Rd>, <Rn>, #<imm3> // (Inside IT block)
```

```
SUBS{<q>} <Rd>, <Rn>, #<imm3> // (Outside IT block)
```

```
d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); 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									

T2

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)
 d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); 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

SUB (S == 0)

SUB<c>.W {<Rd>,} <Rn>, #<const> // (Inside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)
 SUB{<c>}{<q>} {<Rd>,} <Rn>, #<const>

SUBS (S == 1 && Rd != 1111)

SUBS.W {<Rd>,} <Rn>, #<const> // (Outside IT block, and <Rd>, <Rn>, <const> can be represented in T1 or T2)
 SUBS{<c>}{<q>} {<Rd>,} <Rn>, #<const>

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = T32ExpandImm(i:imm3:imm8);
if (d == 15 && !setflags) || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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

T4

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)

```
if Rn == '1111' then SEE "ADR";
if Rn == '1101' then SEE "SUB (SP minus immediate)";
d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
if d == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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							

T5 (!{Rn == 1110 && imm8 == 00000000})

SUBS{<c>}{<q>} PC, LR, #<imm8>

```
if Rn == '1110' && IsZero(imm8) then SEE "ERET";
d = 15; n = UInt(Rn); setflags = TRUE; 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();
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			styp	0	Rm						
cond								Rn																							

SUB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX
```

SUB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

```
SUB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

SUBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

```
SUBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, <shift> #<amount>}
```

```
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(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				

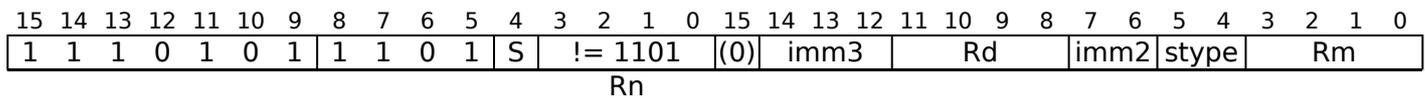
T1

```
SUB<c>{<q>} <Rd>, <Rn>, <Rm> // (Inside IT block)
```

```
SUBS{<q>} {<Rd>}, <Rn>, <Rm> // (Outside IT block)
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTypE_LSL, 0);
```

T2



SUB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

SUB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

SUB, shift or rotate by value (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>}

SUBS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

SUBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, RRX

SUBS, shift or rotate by value (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>}

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
if Rn == '1101' then SEE "SUB (SP minus register)";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || n == 15 || m == 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> 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 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 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 first general-purpose source register, encoded in the "Rn" field. The PC can be used, but this is deprecated. If the SP is used, see [SUB \(SP minus register\)](#).

For encoding T1: 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 "Rn" field. If the SP is used, see [SUB \(SP minus register\)](#).

<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 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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Flag setting (S == 1)

```
SUBS{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, <shift> <Rs>
```

Not flag setting (S == 0)

```
SUB{<c>}{<q>} {<Rd>}, <Rn>, <Rm>, <shift> <Rs>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); 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();
    shift_n = UInt(R[s]<7:0>);
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																																				

SUB (S == 0)

```
SUB{<c>}{<q>} {<Rd>}, SP, #<const>
```

SUBS (S == 1)

```
SUBS{<c>}{<q>} {<Rd>}, SP, #<const>
```

```
d = UInt(Rd); setflags = (S == '1'); 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						

T1

```
SUB{<c>}{<q>} {SP}, SP, #<imm7>
```

```
d = 13; setflags = FALSE; 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								

SUB (S == 0)

```
SUB{<c>}.W {<Rd>}, SP, #<const> // (<Rd>, <const> can be represented in T1)
```

```
SUB{<c>}{<q>} {<Rd>}, SP, #<const>
```

SUBS (S == 1 && Rd != 1111)

```
SUBS{<c>}{<q>} {<Rd>}, SP, #<const>
```

```
if Rd == '1111' && S == '1' then SEE "CMP (immediate)";
d = UInt(Rd); setflags = (S == '1'); 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			Rd			imm8								

T3

```
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)
```

```
d = UInt(Rd); setflags = FALSE; 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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

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:

- 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 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 [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();
  (result, nzcvc) = AddWithCarry(SP, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			styp	0	Rm				
cond																															

SUB, rotate right with extend (S == 0 && imm5 == 00000 && stype == 11)

SUB{<c>}{<q>} {<Rd>}, SP, <Rm> , RRX

SUB, shift or rotate by value (S == 0 && !(imm5 == 00000 && stype == 11))

SUB{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

SUBS, rotate right with extend (S == 1 && imm5 == 00000 && stype == 11)

SUBS{<c>}{<q>} {<Rd>}, SP, <Rm> , RRX

SUBS, shift or rotate by value (S == 1 && !(imm5 == 00000 && stype == 11))

SUBS{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}

```
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(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		styp	Rm				

SUB, rotate right with extend (S == 0 && imm3 == 000 && imm2 == 00 && stype == 11)

```
SUB{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX
```

SUB, shift or rotate by value (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>}
```

SUBS, rotate right with extend (S == 1 && imm3 == 000 && Rd != 1111 && imm2 == 00 && stype == 11)

```
SUBS{<c>}{<q>} {<Rd>}, SP, <Rm>, RRX
```

SUBS, shift or rotate by value (S == 1 && !(imm3 == 000 && imm2 == 00 && stype == 11) && Rd != 1111)

```
SUBS{<c>}{<q>} {<Rd>}, SP, <Rm> {, <shift> #<amount>}
```

```
if Rd == '1111' && S == '1' then SEE "CMP (register)";
d = UInt(Rd); m = UInt(Rm); setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);
if (d == 15 && !setflags) || m == 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> 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();
    shifted = Shift(R[m], shift_t, shift_n, PSTATE.C);
    (result, nzcvc) = AddWithCarry(SP, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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:

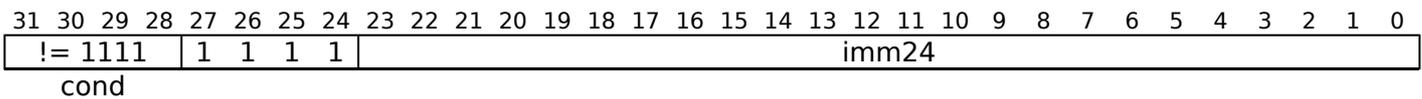
- If the SVC is executed in Hyp mode.
- If *HCR.TGE* is set to 1, and the SVC is executed in Non-secure User mode. For more information, see [Supervisor Call exception, when HCR.TGE is set to 1](#)

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

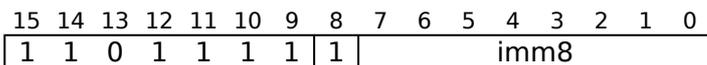


A1

SVC{<c>}{<q>} {#}<imm>

```
imm32 = ZeroExtend(imm24, 32);
```

T1



T1

SVC{<c>}{<q>} {#}<imm>

```
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

SXTAB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

SXTAB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTB";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

SXTAB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

SXTAB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTB16";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

SXTAH{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

SXTAH{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "SXTH";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
SXTB{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); 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		

T1

```
SXTB{<c>}{<q>} {<Rd>}, <Rm>
```

```
d = UInt(Rd); m = UInt(Rm); 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	1	Rd			1	(0)	rotate	Rm				

T2

```
SXTB{<c>}.W {<Rd>}, <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
SXTB{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');  
if d == 15 || m == 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.
- <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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

SXTB16

Signed Extend Byte 16 extracts two 8-bit values from a register, sign-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	0	0	0	1	1	1	1	Rd				rotate	(0)	(0)	0	1	1	1	Rm				
cond																															

A1

SXTB16{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); 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				

T1

SXTB16{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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.
- <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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
SXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); 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	

T1

```
SXTH{<c>}{<q>} {<Rd>}, <Rm>
```

```
d = UInt(Rd); m = UInt(Rm); 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	1	Rd				1	(0)	rotate	Rm			

T2

```
SXTH{<c>}.W {<Rd>}, <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
SXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');  
if d == 15 || m == 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.
- <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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TBB, TBH

Table Branch Byte or Halfword causes a PC-relative forward branch using a table of single byte or halfword offsets. A base register provides a pointer to the table, and a second register supplies an index into the table. The branch length is twice the value returned from the table.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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			

Byte (H == 0)

```
TBB{<c>}{<q>} [<Rn>, <Rm>] // (Outside or last in IT block)
```

Halfword (H == 1)

```
TBH{<c>}{<q>} [<Rn>, <Rm>, LSL #1] // (Outside or last in IT block)
```

```
n = UInt(Rn); m = UInt(Rm); is_tbh = (H == '1');  
if m == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13  
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(PC + 2*halfwords, BranchType_INDIR);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

TEQ{<c>}{<q>} <Rn>, #<const>

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	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							

T1

TEQ{<c>}{<q>} <Rn>, #<const>

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 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](#).

<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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Rotate right with extend (imm5 == 00000 && stype == 11)

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

```
n = UInt(Rn); m = UInt(Rm);  
(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				

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

```
TEQ{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm3 == 000 && imm2 == 00 && stype == 11))

```
TEQ{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

```
n = UInt(Rn); m = UInt(Rm);  
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);  
if n == 15 || m == 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](#).

<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();
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TEQ (register-shifted register)

Test Equivalence (register-shifted register) performs a bitwise exclusive OR operation on a register value and a register-shifted register value. It updates the condition flags based on the result, and discards the result.

A1

31	30	29	28	27	26	25	24	23	22	21	20	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																															

A1

TEQ{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

```
n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
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();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. 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	
!=	1	1	1	1	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																																

A1

TSB{<c>}{<q>} CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
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

T1

TSB{<c>}{<q>} CSYNC

```
if !HaveSelfHostedTrace() then EndOfInstruction(); // Instruction executes as NOP
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

TST{<c>}{<q>} <Rn>, #<const>

```
n = UInt(Rn);
(imm32, carry) = A32ExpandImm_C(imm12, PSTATE.C);
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	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							

T1

TST{<c>}{<q>} <Rn>, #<const>

```
n = UInt(Rn);
(imm32, carry) = T32ExpandImm_C(i:imm3:imm8, PSTATE.C);
if n == 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](#).

<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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Rotate right with extend (imm5 == 00000 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(imm5 == 00000 && stype == 11))

```
TST{<c>}{<q>} <Rn>, <Rm> {, <shift> #<amount>}
```

```
n = UInt(Rn); m = UInt(Rm);  
(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	

T1

```
TST{<c>}{<q>} <Rn>, <Rm>
```

```
n = UInt(Rn); m = UInt(Rm);  
(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	1	Rn				(0)	imm3			1	1	1	1	imm2		stype	Rm				

Rotate right with extend (imm3 == 000 && imm2 == 00 && stype == 11)

```
TST{<c>}{<q>} <Rn>, <Rm>, RRX
```

Shift or rotate by value (!(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>}
```

```
n = UInt(Rn); m = UInt(Rm);  
(shift_t, shift_n) = DecodeImmShift(stype, imm3:imm2);  
if n == 15 || m == 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](#).

<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();
    (shifted, carry) = Shift\_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

TST{<c>}{<q>} <Rn>, <Rm>, <type> <Rs>

```
n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
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();
    shift_n = UInt(R[s]<7:0>);
    (shifted, carry) = Shift_C(R[m], shift_t, shift_n, PSTATE.C);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
UADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); 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	Rn				1	1	1	1	Rd				0	1	0	0	Rm						

T1

```
UADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);  
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);  
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(lsb); widthminus1 = UInt(widthm1);
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			

T1

UBFX{<c>}{<q>} <Rd>, <Rn>, #<lsb>, #<width>

```
d = UInt(Rd); n = UInt(Rn);
lsbit = UInt(imm3:imm2); widthminus1 = UInt(widthm1);
msbit = lsbit + widthminus1;
if d == 15 || n == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Rd> Is the general-purpose destination register, encoded in the "Rd" field.

- <Rn> Is the general-purpose source register, encoded in the "Rn" field.
- <lsb> For 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UDF

Permanently Undefined generates an Undefined Instruction exception.

The encodings for UDF used in this section are defined as permanently UNDEFINED. However:

- With the T32 instruction set, Arm deprecates using the UDF instruction in an IT block.
- In the A32 instruction set, UDF is not conditional.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	0	1	1	1	1	1	1	1	imm12												1	1	1	1	imm4			
cond																															

A1

```
UDF{<c>}{<q>} {#}<imm>
```

```
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							

T1

```
UDF{<c>}{<q>} {#}<imm>
```

```
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	1	imm4				1	0	1	0	imm12											

T2

```
UDF{<c>}.W {#}<imm> // (<imm> can be represented in T1)
```

```
UDF{<c>}{<q>} {#}<imm>
```

```
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																							

A1

UDIV{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

UDIV{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
if d == 15 || n == 15 || m == 15 || a != 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

CONSTRAINED UNPREDICTABLE behavior

If Ra != '1111', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as NOP.
- The instruction executes as described, with no change to its behavior and no additional side effects.
- The instruction performs a divide and the register specified by Ra becomes UNKNOWN.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<Rd> Is the general-purpose destination register, encoded in the "Rd" field.

<Rn> Is the first general-purpose source register holding the dividend, encoded in the "Rn" field.

<Rm> Is the second general-purpose source register holding the divisor, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
integer result;
if UInt(R[m]) == 0 then
    result = 0;
else
    result = RoundTowardsZero(Real(UInt(R[n])) / Real(UInt(R[m])));
R[d] = result<31:0>;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UHADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UHADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
UHADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

```
UHADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>
```

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);  
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations();  
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);  
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);  
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);  
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UHASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UHASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UHSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UHSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UHSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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	Rn				1	1	1	1	Rd				0	1	1	0	Rm						

T1

UHSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UHSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UHSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); 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				Rn				RdLo				RdHi				0 1 1 0				Rm			

T1

UMAAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); 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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<RdLo> Is the general-purpose source register holding the first addend and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.

- <RdHi> Is the general-purpose source register holding the second addend and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
- <Rn> Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMLAL, UMLALS

Unsigned Multiply Accumulate Long multiplies two unsigned 32-bit values to produce a 64-bit value, and accumulates this with a 64-bit value.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 0 0 0				1 0 1		S	RdHi				RdLo				Rm				1 0 0 1				Rn				
cond																															

Flag setting (S == 1)

```
UMLALS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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				Rn				RdLo				RdHi				0 0 0 0				Rm			

T1

```
UMLAL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<RdLo>	Is the general-purpose source register holding the lower 32 bits of the addend, and the destination register for the lower 32 bits of the result, encoded in the "RdLo" field.
<RdHi>	Is the general-purpose source register holding the upper 32 bits of the addend, and the destination register for the upper 32 bits of the result, encoded in the "RdHi" field.
<Rn>	Is the first general-purpose source register holding the multiplicand, encoded in the "Rn" field.
<Rm>	Is the second general-purpose source register holding the multiplier, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UMULL, UMULLS

Unsigned Multiply Long multiplies two 32-bit unsigned values to produce a 64-bit result.

In A32 instructions, the condition flags can optionally be updated based on the result. Use of this option adversely affects performance on many implementations.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				0 0 0 0				1 0 0				S	RdHi				RdLo				Rm				1 0 0 1				Rn			
cond																																

Flag setting (S == 1)

```
UMULLS{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

Not flag setting (S == 0)

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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			

T1

```
UMULL{<c>}{<q>} <RdLo>, <RdHi>, <Rn>, <Rm>
```

```
dLo = UInt(RdLo); dHi = UInt(RdHi); n = UInt(Rn); m = UInt(Rm); 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 [Standard assembler syntax fields](#).

<q>	See Standard assembler syntax fields .
<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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

UQADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQADD16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<15:0>) + UInt(R[m]<15:0>);
    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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

UQADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQADD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum1 = UInt(R[n]<7:0>) + UInt(R[m]<7:0>);
    sum2 = UInt(R[n]<15:8>) + UInt(R[m]<15:8>);
    sum3 = UInt(R[n]<23:16>) + UInt(R[m]<23:16>);
    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);
```

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																															

A1

UQASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQASX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
diff = UInt(R[n]<15:0>) - UInt(R[m]<31:16>);
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																															

A1

UQSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQSAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

UQSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQSUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    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);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

A1

UQSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

UQSUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    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);
```

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																															

A1

USAD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

USAD8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    absdiff1 = Abs(UInt(R[n]<7:0>) - UInt(R[m]<7:0>));
    absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
    absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
    absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
    result = absdiff1 + absdiff2 + absdiff3 + absdiff4;
    R[d] = result<31:0>;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USADA8

Unsigned Sum of Absolute Differences and Accumulate performs four unsigned 8-bit subtractions, and adds the absolute values of the differences to a 32-bit accumulate operand.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0 1 1 1 1 0 0 0				Rd				!= 1111				Rm				0 0 0 1				Rn							
cond								Ra																							

A1

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

USADA8{<c>}{<q>} <Rd>, <Rn>, <Rm>, <Ra>

```
if Ra == '1111' then SEE "USAD8";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); a = UInt(Ra);
if d == 15 || n == 15 || m == 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, 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();
absdiff1 = Abs(UInt(R[n]<7:0>) - UInt(R[m]<7:0>));
absdiff2 = Abs(UInt(R[n]<15:8>) - UInt(R[m]<15:8>));
absdiff3 = Abs(UInt(R[n]<23:16>) - UInt(R[m]<23:16>));
absdiff4 = Abs(UInt(R[n]<31:24>) - UInt(R[m]<31:24>));
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Arithmetic shift right (sh == 1)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

```
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm);
(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					

Arithmetic shift right (sh == 1 && !(imm3 == 000 && imm2 == 00))

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn>, ASR #<amount>

Logical shift left (sh == 0)

USAT{<c>}{<q>} <Rd>, #<imm>, <Rn> {, LSL #<amount>}

```
if sh == '1' && (imm3:imm2) == '00000' then SEE "USAT16";
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm);
(shift_t, shift_n) = DecodeImmShift(sh:'0', imm3:imm2);
if d == 15 || n == 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.
- <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 T1: 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();
  operand = Shift(R[n], shift_t, shift_n, PSTATE.C); // PSTATE.C ignored
  (result, sat) = UnsignedSatQ(SInt(operand), saturate_to);
  R[d] = ZeroExtend(result, 32);
  if sat then
    PSTATE.Q = '1';
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

USAT16

Unsigned Saturate 16 saturates two signed 16-bit values to a selected unsigned range.

This instruction sets *PSTATE.Q* to 1 if the operation saturates.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	1	0	sat_imm				Rd				(1)	(1)	(1)	(1)	0	0	1	1	Rn			
cond																															

A1

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

USAT16{<c>}{<q>} <Rd>, #<imm>, <Rn>

```
d = UInt(Rd); n = UInt(Rn); saturate_to = UInt(sat_imm);
if d == 15 || n == 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.
- <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();
    (result1, sat1) = UnsignedSatQ(SInt(R[n]<15:0>), saturate_to);
    (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																															

A1

USAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

USAX{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    sum = UInt(R[n]<15:0>) + UInt(R[m]<31:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

USUB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<15:0>) - UInt(R[m]<15:0>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

USUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); 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			

T1

USUB8{<c>}{<q>} {<Rd>}, <Rn>, <Rm>

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm);
if d == 15 || n == 15 || m == 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, encoded in the "Rn" field.
- <Rm> Is the second general-purpose source register, encoded in the "Rm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    diff1 = UInt(R[n]<7:0>) - UInt(R[m]<7:0>);
    diff2 = UInt(R[n]<15:8>) - UInt(R[m]<15:8>);
    diff3 = UInt(R[n]<23:16>) - UInt(R[m]<23:16>);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

UXTAB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

UXTAB{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTB";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

UXTAB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

UXTAB16{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTB16";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

UXTAH{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); 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																															

T1

UXTAH{<c>}{<q>} {<Rd>}, <Rn>, <Rm> {, ROR #<amount>}

```
if Rn == '1111' then SEE "UXTH";
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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, 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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

```
UXTB{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); 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		

T1

```
UXTB{<c>}{<q>} {<Rd>}, <Rm>
```

```
d = UInt(Rd); m = UInt(Rm); 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	1	Rd			1	(0)	rotate	Rm				

T2

```
UXTB{<c>}.W {<Rd>}, <Rm> // (<Rd>, <Rm> can be represented in T1)
```

```
UXTB{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}
```

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');  
if d == 15 || m == 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.

<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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

UXTB16

Unsigned Extend Byte 16 extracts two 8-bit values from a register, zero-extends them to 16 bits each, and writes the results to the destination register. The instruction can specify a rotation by 0, 8, 16, or 24 bits before extracting the 8-bit values.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	1	1	0	1	1	0	0	1	1	1	1	Rd				rotate	(0)	(0)	0	1	1	1	Rm				

cond

A1

UXTB16{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); 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				

T1

UXTB16{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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.
- <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();
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	Rd				rotate	(0)	(0)	0	1	1	1	Rm								
cond																															

A1

UXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); 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		

T1

UXTH{<c>}{<q>} {<Rd>}, <Rm>

```
d = UInt(Rd); m = UInt(Rm); 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				

T2

UXTH{<c>}.W {<Rd>}, <Rm> // (<Rd>, <Rm> can be represented in T1)

UXTH{<c>}{<q>} {<Rd>}, <Rm> {, ROR #<amount>}

```
d = UInt(Rd); m = UInt(Rm); rotation = UInt(rotate:'000');
if d == 15 || m == 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.
- <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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

WFE

Wait For Event is a hint instruction that indicates that the PE can enter a low-power state and remain there until a wakeup event occurs. Wakeup events include the event signaled as a result of executing the SEV instruction on any PE in the multiprocessor system. For more information, see [Wait For Event and Send Event](#).

As described in [Wait For Event and Send Event](#), the execution of a WFE instruction that would otherwise cause entry to a low-power state can be trapped to a higher Exception level, see:

- [Traps to Undefined mode of PL0 execution of WFE and WFI instructions](#).
- [Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions](#).
- [Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor 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	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																																

A1

WFE{<c>}{<q>}

// 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

T1

WFE{<c>}{<q>}

// 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	0

T2

WFE{<c>}.W

// 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 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\(EL3, WfxType\_WFE\);
    integer localtimeout = 1 << 64; // No local timeout event is generated
    WaitForEvent\(localtimeout\);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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:

- [Traps to Undefined mode of PL0 execution of WFE and WFI instructions](#).
- [Traps to Hyp mode of Non-secure EL0 and EL1 execution of WFE and WFI instructions](#).
- [Traps to Monitor mode of the execution of WFE and WFI instructions in modes other than Monitor 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	1	0	0	1	0	0	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	1	1	
cond																															

A1

```
WFI{<c>}{<q>}
```

```
// 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

T1

```
WFI{<c>}{<q>}
```

```
// 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

T2

```
WFI{<c>}.W
```

```
// 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 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\(EL3, WfxType\_WFI\);
        integer localtimeout = 1 << 64; // No local timeout event is generated
        WaitForInterrupt(localtimeout);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	0	0	0	0	0	1

cond

A1

```
YIELD{<c>}{<q>}
```

```
// 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

T1

```
YIELD{<c>}{<q>}
```

```
// 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

T2

```
YIELD{<c>}.W
```

```
// 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\(\);
```

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.

[VCMPE](#): 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 \(BFloat16, by scalar\)](#): BFloat16 floating-point widening multiply-add long (by scalar).

[VFMA, VFMA \(BFloat16, vector\)](#): BFloat16 floating-point widening multiply-add long (vector).

[VFMAL \(by scalar\)](#): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

[VFMAL \(vector\)](#): Vector Floating-point Multiply-Add Long to accumulator (vector).

[VFMS](#): Vector Fused Multiply Subtract.

[VFMSL \(by scalar\)](#): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

[VFMSL \(vector\)](#): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

[VFNMA](#): Vector Fused Negate Multiply Accumulate.

[VFNMS](#): Vector Fused Negate Multiply Subtract.

[VHADD](#): Vector Halving Add.

[VHSUB](#): Vector Halving Subtract.

[VINS](#): Vector move Insertion.

[VJCVT](#): Javascript Convert to signed fixed-point, rounding toward Zero.

[VLD1 \(multiple single elements\)](#): Load multiple single 1-element structures to one, two, three, or four registers.

[VLD1 \(single element to all lanes\)](#): Load single 1-element structure and replicate to all lanes of one register.

[VLD1 \(single element to one lane\)](#): Load single 1-element structure to one lane of one register.

[VLD2 \(multiple 2-element structures\)](#): Load multiple 2-element structures to two or four registers.

[VLD2 \(single 2-element structure to all lanes\)](#): Load single 2-element structure and replicate to all lanes of two registers.

- [VLD2 \(single 2-element structure to one lane\)](#): Load single 2-element structure to one lane of two registers.
- [VLD3 \(multiple 3-element structures\)](#): Load multiple 3-element structures to three registers.
- [VLD3 \(single 3-element structure to all lanes\)](#): Load single 3-element structure and replicate to all lanes of three registers.
- [VLD3 \(single 3-element structure to one lane\)](#): Load single 3-element structure to one lane of three registers.
- [VLD4 \(multiple 4-element structures\)](#): Load multiple 4-element structures to four registers.
- [VLD4 \(single 4-element structure to all lanes\)](#): Load single 4-element structure and replicate to all lanes of four registers.
- [VLD4 \(single 4-element structure to one lane\)](#): Load single 4-element structure to one lane of four registers.
- [VLDM, VLDMDB, VLDMIA](#): Load Multiple SIMD&FP registers.
- [VLDL \(immediate\)](#): Load SIMD&FP register (immediate).
- [VLDL \(literal\)](#): 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).
- [VMLS \(by scalar\)](#): Vector Multiply Subtract Long (by scalar).
- [VMLS \(integer\)](#): 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.

[VQRDLAH](#): 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.

[VRSQRT](#): Vector Reciprocal Square Root Estimate.

[VRSQRTS](#): Vector Reciprocal Square Root Step.

[VRSRA](#): 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.

[VSR](#): 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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				

A1

AESD.<dt> <Qd>, <Qm>

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	0	0		Vd		0	0	1	1	0	1	M	0		Vm				

T1

AESD.<dt> <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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();
    op1 = Q[d>>1]; 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

AESE

AESE single round encryption.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

A1

AESE.<dt> <Qd>, <Qm>

```
if !HaveAESEExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	0	0		Vd				0	0	1	1	0	0	M	0		Vm		

T1

AESE.<dt> <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESEExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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();
    op1 = Q[d>>1]; 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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							

A1

AESIMC.<dt> <Qd>, <Qm>

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	0	0	Vd	0	0	1	1	1	1	M	0	Vm							

T1

AESIMC.<dt> <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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				

A1

AESMC.<dt> <Qd>, <Qm>

```
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	0	0		Vd		0	0	1	1	1	0	M	0		Vm				

T1

AESMC.<dt> <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAESExt() then UNDEFINED;
if size != '00' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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>>1] = AESMixColumns(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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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>			

Decrement Before (P == 1 && U == 0 && W == 1)

FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

```

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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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>			

Decrement Before (P == 1 && U == 0 && W == 1)

FLDMDBX{<c>}{<q>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

FLDMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

```
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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.
!	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
            word1 = MemA[address,4]; 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(AccType_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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>			

Decrement Before (P == 1 && U == 0 && W == 1)

FSTMDBX{<c>}{<q>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

```

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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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>			

Decrement Before (P == 1 && U == 0 && W == 1)

FSTMDBX{<c>}{<q>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

FSTMIAX{<c>}{<q>} <Rn>{!}, <dreglist>

```
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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Rn>	Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.
!	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.
      MemA[address,4] = if BigEndian(AccType\_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType\_ATOMIC) then D[d+r]<31:0> else 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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					

A1

SHA1C.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	0	0		Vn		Vd		1	1	0	0	N	Q	M	0		Vm					

T1

SHA1C.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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
    W = Q[m>>1];
    for e = 0 to 3
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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				

A1

SHA1H.32 <Qd>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	0	1		Vd		0	0	1	0	1	1	M	0		Vm				

T1

SHA1H.32 <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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.

SHA1M

SHA1 hash update (majority).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

A1

SHA1M.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	1	0		Vn		Vd					1	1	0	0	N	Q	M	0		Vm		

T1

SHA1M.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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
    W = Q[m>>1];
    for e = 0 to 3
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

A1

SHA1P.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	0	1		Vn		Vd					1	1	0	0	N	Q	M	0		Vm		

T1

SHA1P.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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
    W = Q[m>>1];
    for e = 0 to 3
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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			

A1

SHA1SU0.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	1	1	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

SHA1SU0.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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();
    op1 = Q[d>>1]; op2 = Q[n>>1]; 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.

SHA1SU1

SHA1 schedule update 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
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		

A1

SHA1SU1.32 <Qd>, <Qm>

```
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	1	1	0	M	0		Vm		

T1

SHA1SU1.32 <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA1Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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();
    X = Q[d>>1]; Y = Q[m>>1];
    T = X EOR LSR(Y, 32);
    W0 = ROL(T<31:0>, 1);
    W1 = ROL(T<63:32>, 1);
    W2 = ROL(T<95:64>, 1);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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					

A1

SHA256H.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	0		Vn		Vd		1	1	0	0	N	Q	M	0		Vm					

T1

SHA256H.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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]; W = Q[m>>1]; 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.

SHA256H2

SHA256 hash update part 2.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

A1

SHA256H2.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	1		Vn		Vd					1	1	0	0	N	Q	M	0		Vm		

T1

SHA256H2.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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[n>>1]; Y = Q[d>>1]; W = Q[m>>1]; 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.

SHA256SU0

SHA256 schedule update 0.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

A1

SHA256SU0.32 <Qd>, <Qm>

```
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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	1	1	1	M	0		Vm		

T1

SHA256SU0.32 <Qd>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if size != '10' then UNDEFINED;
if Vd<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); 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();
  X = Q[d>>1]; Y = Q[m>>1];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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			

A1

SHA256SU1.32 <Qd>, <Qn>, <Qm>

```
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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	0	Vn				Vd				1	1	0	0	N	Q	M	0	Vm			

T1

SHA256SU1.32 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveSHA256Ext() then UNDEFINED;
if Q != '1' then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
  X = Q\[d>>1\]; Y = Q\[n>>1\]; Z = Q\[m>>1\];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	

64-bit SIMD vector (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VABA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

A1

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																															

T1

VABAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	

64-bit SIMD vector (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
      Elem[D[d+r],e,esize] = FPAbs(FPSub(op1,op2,StandardFPSCRValue()));
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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		

64-bit SIMD vector (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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		

64-bit SIMD vector (Q == 0)

VABD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VABD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

T1

VABDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize];
            op2 = Elem[Din[m+r],e,esize];
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (Q == 0)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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

Half-precision scalar (size == 01) (FEAT_FP16)

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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				

64-bit SIMD vector (Q == 0)

VABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VABS{<c>}{<q>}.<dt> <Qd>, <Qm>

```

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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					

Half-precision scalar (size == 01) (FEAT_FP16)

VABS{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VABS{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VABS{<c>}{<q>}.F64 <Dd>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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
        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]);
                else
                    result = Abs(SInt(Elem[D[m+r],e,esize]));
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else
        // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPAbs(S[m]<15:0>);
            when 32 S[d] = FPAbs(S[m]);
            when 64 D[d] = FPAbs(D[m]);
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

64-bit SIMD vector (Q == 0)

```
VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																															

64-bit SIMD vector (Q == 0)

VACGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VACGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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();
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = FPAbs(Elem[D[n+r],e,esize]); op2 = FPAbs(Elem[D[m+r],e,esize]);
      boolean test_passed;
      if or_equal then
        test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
      else
        test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
      Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

```


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

64-bit SIMD vector (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
or_equal = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VACGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VACGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
or_equal = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = FPAbs(Elem[D[n+r],e,esize]); op2 = FPAbs(Elem[D[m+r],e,esize]);
            boolean test_passed;
            if or_equal then
                test_passed = FPCompareGE(op1, op2, StandardFPSCRValue());
            else
                test_passed = FPCompareGT(op1, op2, StandardFPSCRValue());
            Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

```


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 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																															

64-bit SIMD vector (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VACLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 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																															

64-bit SIMD vector (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VACLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VACGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VADD (floating-point)

Vector Add (floating-point) adds corresponding elements in two vectors, and places the results in the destination vector.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

```
VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																															

Half-precision scalar (size == 01)
(FEAT_FP16)

VADD{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VADD{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VADD{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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					

64-bit SIMD vector (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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						

Half-precision scalar (size == 01) (FEAT_FP16)

VADD{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VADD{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VADD{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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
        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],
                    StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPAdd(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPAdd(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPAdd(D[n], D[m], FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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	size			Vn		Vd					1	0	0	0	N	Q	M	0			Vm	

64-bit SIMD vector (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

A1

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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	0	0	N	0	M	0	Vm										

size

T1

VADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

A1

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); 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	!= 11				Vn					Vd			0	0	0	0	N	0	M	0		Vm
size											op																				

T1

VADDL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); 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 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);
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

A1

VADDW{<c>}{<q>}.<dt> {<Qd>}, {<Qn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); 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	!= 11	Vn					Vd					0	0	0	1	N	0	M	0	Vm		
size											op																				

T1

VADDW{<c>}{<q>}.<dt> {<Qd>}, {<Qn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize; is_vaddw = (op == '1');
d = UInt(D:Vd); n = UInt(N:Vn); 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 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);
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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																															

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>}, {<Dd>}, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #-<imm>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>}, {<Dd>}, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #-<imm>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}.I16 {<Dd>}, <Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I16 <Dd>, #~<imm>

128-bit SIMD vector (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																			

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}.I32 {<Dd>}, <Dd>, #<imm>

is equivalent to

VBIC{<c>}{<q>}.I32 <Dd>, #~<imm>

128-bit SIMD vector (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.

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			

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;  
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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			

64-bit SIMD vector (Q == 0)

VAND{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VAND{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;  
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}.I32 {<Dd>}, {<Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>}, {<Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";  
if Q == '1' && Vd<0> == '1' then UNDEFINED;  
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);  
d = UInt(D:Vd); 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																															

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}.I16 {<Dd>}, {<Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>}, {<Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";  
if Q == '1' && Vd<0> == '1' then UNDEFINED;  
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);  
d = UInt(D:Vd); 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																															

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}.I32 {<Dd>}, <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I32 {<Qd>}, <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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															

64-bit SIMD vector (Q == 0)

```
VBIC{<c>}{<q>}.I16 {<Dd>}, <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VBIC{<c>}{<q>}.I16 {<Qd>}, <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	1		Vn		Vd		0	0	0	1	N	Q	M	1		Vm					

64-bit SIMD vector (Q == 0)

VBIC{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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					

64-bit SIMD vector (Q == 0)

VBIC{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VBIC{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
         $D[d+r] = D[n+r] \text{ AND NOT}(D[m+r]);$ 
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

```
VBIF{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBIF{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VBIT{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VBIT{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VBIT{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VBIT{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VBSL

Vector Bitwise Select sets each bit in the destination to the corresponding bit from the first source operand when the original destination bit was 1, otherwise from the second source operand.

Depending on settings in the *CPACR*, *NSACR*, and *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	0	D	0	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

op

64-bit SIMD vector (Q == 0)

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

```
VBSL{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VBSL{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
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;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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					

64-bit SIMD vector (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
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					

64-bit SIMD vector (Q == 0)

VCADD{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

VCADD{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
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
  operand1 = D[n+r];
  operand2 = D[m+r];
  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]);
        element3 = Elem[operand2,e*2,esize];
      when '1'
        element1 = Elem[operand2,e*2+1,esize];
        element3 = FPNeg(Elem[operand2,e*2,esize]);
  result1 = FPAdd(Elem[operand1,e*2,esize],element1,StandardFPSCRValue());
  result2 = FPAdd(Elem[operand1,e*2+1,esize],element3,StandardFPSCRValue());
  Elem[D[d+r],e*2,esize] = result1;
  Elem[D[d+r],e*2+1,esize] = result2;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	1	Vd	0	F	0	1	0	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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	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
        bits(esize) zero = FPZero('0');
        test_passed = FPCompareEQ(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
      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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCEQ (register)

Vector Compare Equal takes each element in a vector, and compares it with the corresponding element of a second vector. If they are equal, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	size				Vn					Vd			1	0	0	0	N	Q	M	1		Vm

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

```
VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
int_operation = FALSE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
int_operation = TRUE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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					

64-bit SIMD vector (Q == 0)

VCEQ{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCEQ{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
int_operation = FALSE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
      op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
      boolean test_passed;
      if int_operation then
        test_passed = (op1 == op2);
      else
        test_passed = FPCompareEQ(op1, op2, StandardFPSCRValue\(\));
      Elem[D[d+r],e,esize] = if test_passed then Ones(esize) else Zeros(esize);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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										

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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									

64-bit SIMD vector (Q == 0)

```
VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGEType_fp;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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										

64-bit SIMD vector (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGEType_unsigned else VCGEType_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VCGE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGE{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGEType_fp;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[D[n+r],e,esize]; 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, StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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			

64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
        bits(esize) zero = FPZero('0');
        test_passed = FPCompareGT(Elem[D[m+r],e,esize], zero, StandardFPSCRValue());
      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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCGT (register)

Vector Compare Greater Than takes each element in a vector, and compares it with the corresponding element of a second vector. If the first is greater than the second, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers, unsigned integers, or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instruction [VCLT \(register\)](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size				Vn					Vd		0	0	1	1	N	Q	M	0		Vm	

64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype\_unsigned else VCGTtype\_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

```
VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
vtype = VCGTtype\_fp;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
vtype = if U == '1' then VCGTtype_unsigned else VCGTtype_signed;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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					

64-bit SIMD vector (Q == 0)

VCGT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VCGT{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
vtype = VCGTtype_fp;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[D[n+r],e,esize]; 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, StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	1	Vd	0	F	0	1	1	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCLE{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLE{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGE(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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										

64-bit SIMD vector (Q == 0)

`VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>`

128-bit SIMD vector (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									

64-bit SIMD vector (Q == 0)

`VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>`

is equivalent to

`VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>`

128-bit SIMD vector (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										

64-bit SIMD vector (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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					

64-bit SIMD vector (Q == 0)

VCLE{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGE{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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							

64-bit SIMD vector (Q == 0)

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCLS{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLS{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLT (immediate #0)

Vector Compare Less Than Zero takes each element in a vector, and compares it with zero. If it is less than zero, the corresponding element in the destination vector is set to all ones. Otherwise, it is set to all zeros.

The operand vector elements are the same type, and are signed integers or floating-point numbers. The result vector elements are fields the same size as the operand vector elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	1	Vd	0	F	1	0	0	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCLT{<c>}{<q>}.<dt> {<Dd>}, <Dm>, #0
```

128-bit SIMD vector (Q == 1)

```
VCLT{<c>}{<q>}.<dt> {<Qd>}, <Qm>, #0
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
                bits(esize) zero = FPZero('0');
                test_passed = FPCompareGT(zero, Elem[D[m+r],e,esize], StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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										

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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									

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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										

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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					

64-bit SIMD vector (Q == 0)

VCLT{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

is equivalent to

VCGT{<c>}{<q>}.<dt> <Dd>, <Dm>, <Dn>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCLZ

Vector Count Leading Zeros counts the number of consecutive zeros, starting from the most significant bit, in each element in a vector, and places the results in a second vector.

The operand vector elements can be any one of 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The result vector elements are the same data type as the operand vector elements.

Depending on settings in the *CPACR*, *NSACR*, and *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							

64-bit SIMD vector (Q == 0)

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCLZ{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VCLZ{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCMLA

Vector Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following 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			

64-bit SIMD vector (Q == 0)

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

```
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
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			

64-bit SIMD vector (Q == 0)

VCMLA{<q>}.<dt> <Dd>, <Dn>, <Dm>, #<rotate>

128-bit SIMD vector (Q == 1)

VCMLA{<q>}.<dt> <Qd>, <Qn>, <Qm>, #<rotate>

```
if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
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();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = D[m+r];
  operand3 = D[d+r];
  for e = 0 to (elements DIV 2)-1
    bits(usize) element1;
    bits(usize) element2;
    bits(usize) element3;
    bits(usize) element4;
    case rot of
      when '00'
        element1 = Elem[operand2,e*2,usize];
        element2 = Elem[operand1,e*2,usize];
        element3 = Elem[operand2,e*2+1,usize];
        element4 = Elem[operand1,e*2,usize];
      when '01'
        element1 = FPNeg(Elem[operand2,e*2+1,usize]);
        element2 = Elem[operand1,e*2+1,usize];
        element3 = Elem[operand2,e*2,usize];
        element4 = Elem[operand1,e*2+1,usize];
      when '10'
        element1 = FPNeg(Elem[operand2,e*2,usize]);
        element2 = Elem[operand1,e*2,usize];
        element3 = FPNeg(Elem[operand2,e*2+1,usize]);
        element4 = Elem[operand1,e*2,usize];
      when '11'
        element1 = Elem[operand2,e*2+1,usize];
        element2 = Elem[operand1,e*2+1,usize];
        element3 = FPNeg(Elem[operand2,e*2,usize]);
        element4 = Elem[operand1,e*2+1,usize];
    result1 = FPMulAdd(Elem[operand3,e*2,usize],element2,element1, StandardFPSCRValue());
    result2 = FPMulAdd(Elem[operand3,e*2+1,usize],element4,element3, StandardFPSCRValue());
    Elem[D[d+r],e*2,usize] = result1;
    Elem[D[d+r],e*2+1,usize] = result2;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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				

64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

```

if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
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				

64-bit SIMD vector of half-precision floating-point (S == 0 && Q == 0)

VCMLA{<q>}.F16 <Dd>, <Dn>, <Dm>[<index>], #<rotate>

64-bit SIMD vector of single-precision floating-point (S == 1 && Q == 0)

VCMLA{<q>}.F32 <Dd>, <Dn>, <Dm>[0], #<rotate>

128-bit SIMD vector of half-precision floating-point (S == 0 && Q == 1)

VCMLA{<q>}.F16 <Qd>, <Qn>, <Dm>[<index>], #<rotate>

128-bit SIMD vector of single-precision floating-point (S == 1 && Q == 1)

VCMLA{<q>}.F32 <Qd>, <Qn>, <Dm>[0], #<rotate>

```

if InITBlock() then UNPREDICTABLE;
if !HaveFCADDExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn);
m = if S=='1' then UInt(M:Vm) else UInt(Vm);
esize = 16 << UInt(S);
if !HaveFP16Ext() && esize == 16 then UNDEFINED;
elements = 64 DIV esize;
regs = if Q == '0' then 1 else 2;
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();
for r = 0 to regs-1
  operand1 = D[n+r];
  operand2 = Din[m];
  operand3 = D[d+r];
  for e = 0 to (elements DIV 2)-1
    bits(ysize) element1;
    bits(ysize) element2;
    bits(ysize) element3;
    bits(ysize) element4;
    case rot of
      when '00'
        element1 = Elem[operand2,index*2,ysize];
        element2 = Elem[operand1,e*2,ysize];
        element3 = Elem[operand2,index*2+1,ysize];
        element4 = Elem[operand1,e*2,ysize];
      when '01'
        element1 = FPNeg(Elem[operand2,index*2+1,ysize]);
        element2 = Elem[operand1,e*2+1,ysize];
        element3 = Elem[operand2,index*2,ysize];
        element4 = Elem[operand1,e*2+1,ysize];
      when '10'
        element1 = FPNeg(Elem[operand2,index*2,ysize]);
        element2 = Elem[operand1,e*2,ysize];
        element3 = FPNeg(Elem[operand2,index*2+1,ysize]);
        element4 = Elem[operand1,e*2,ysize];
      when '11'
        element1 = Elem[operand2,index*2+1,ysize];
        element2 = Elem[operand1,e*2+1,ysize];
        element3 = FPNeg(Elem[operand2,index*2,ysize]);
        element4 = Elem[operand1,e*2+1,ysize];
    result1 = FPMulAdd(Elem[operand3,e*2,ysize],element2,element1, StandardFPSCRValue());
    result2 = FPMulAdd(Elem[operand3,e*2+1,ysize],element4,element3,StandardFPSCRValue());
    Elem[D[d+r],e*2,ysize] = result1;
    Elem[D[d+r],e*2+1,ysize] = result2;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																	

Half-precision scalar (size == 01) (FEAT_FP16)

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

```

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)	(0)
cond														E																	

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VCMP{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, #0.0

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd			1	0	size	0	1	M	0	Vm					
																E															

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VCMP{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMP{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMP{<c>}{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	1	Vd		1	0	size	0	1	(0)	0	(0)	(0)	(0)	(0)			
																E															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VCMP{<c>}{<q>}.F16 <Sd>, #0.0
```

Single-precision scalar (size == 10)

```
VCMP{<c>}{<q>}.F32 <Sd>, #0.0
```

Double-precision scalar (size == 11)

```
VCMP{<c>}{<q>}.F64 <Dd>, #0.0
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    bits(4) nzcvc;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcvc = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR[]);
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcvc = FPCompare(S[d], op32, quiet_nan_exc, FPSCR[]);
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcvc = FPCompare(D[d], op64, quiet_nan_exc, FPSCR[]);

    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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
!= 1111				1	1	1	0	1	D	1	1	0	1	0	0	Vd				1	0	size	1	1	M	0	Vm				
cond											E																				

Half-precision scalar (size == 01) (FEAT_FP16)

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

```

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)	(0)
cond											E																				

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VCMPE{<c>}{<q>}.F16 <Sd>, #0.0

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, #0.0

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, #0.0

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && cond != '1110', then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
1	1	1	0	1	1	1	0	1	D	1	1	0	1	0	0	Vd			1	0	size	1	1	M	0	Vm									
																E																			

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VCMPE{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCMPE{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCMPE{<c>}{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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)			
																E															

Half-precision scalar (size == 01) (FEAT_FP16)

`VCMPE{<c>}{<q>}.F16 <Sd>, #0.0`

Single-precision scalar (size == 10)

`VCMPE{<c>}{<q>}.F32 <Sd>, #0.0`

Double-precision scalar (size == 11)

`VCMPE{<c>}{<q>}.F64 <Dd>, #0.0`

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = TRUE;
integer esize;
integer d;
case size of
  when '01' esize = 16; d = UInt(Vd:D);
  when '10' esize = 32; d = UInt(Vd:D);
  when '11' esize = 64; d = UInt(D:Vd);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    bits(4) nzcvc;
    case esize of
        when 16
            bits(16) op16 = if with_zero then FPZero('0') else S[m]<15:0>;
            nzcvc = FPCompare(S[d]<15:0>, op16, quiet_nan_exc, FPSCR());
        when 32
            bits(32) op32 = if with_zero then FPZero('0') else S[m];
            nzcvc = FPCompare(S[d], op32, quiet_nan_exc, FPSCR());
        when 64
            bits(64) op64 = if with_zero then FPZero('0') else D[m];
            nzcvc = FPCompare(D[d], op64, quiet_nan_exc, FPSCR());

    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

64-bit SIMD vector (Q == 0)

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8; elements = 8;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VCNT{<c>}{<q>}.8 <Dd>, <Dm> // (Encoded as Q = 0)
```

128-bit SIMD vector (Q == 1)

```
VCNT{<c>}{<q>}.8 <Qd>, <Qm> // (Encoded as Q = 1)
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8; elements = 8;
d = UInt(D:Vd); m = UInt(M:Vm); 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](#).
- <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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

Single-precision to double-precision (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Double-precision to single-precision (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

```
double_to_single = (size == '11');  
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);  
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															

Single-precision to double-precision (size == 10)

```
VCVT{<c>}{<q>}.F64.F32 <Dd>, <Sm>
```

Double-precision to single-precision (size == 11)

```
VCVT{<c>}{<q>}.F32.F64 <Sd>, <Dm>
```

```
double_to_single = (size == '11');  
d = if double_to_single then UInt(Vd:D) else UInt(D:Vd);  
m = if double_to_single then UInt(M:Vm) else UInt(Vm:M);
```

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <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(); CheckVFPEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[]);
    else
        D[d] = FPConvert(S[m], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and fixed-point, Advanced SIMD)

Vector Convert between floating-point and fixed-point converts each element in a vector from floating-point to fixed-point, or from fixed-point to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to fixed-point operation uses the Round towards Zero rounding mode. The fixed-point to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, and *HCPtr* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd			1	1	op	0	Q	M	1	Vm					

64-bit SIMD vector (imm6 != 000xxx && Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 IN {'10xxxx'} then UNDEFINED;
if imm6 IN {'0xxxxx'} then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1'); frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
integer esize;
integer elements;
case op<1> of
    when '0' esize = 16; elements = 4;
    when '1' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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					

64-bit SIMD vector (imm6 != 000xxx && Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>, #<fbits>

128-bit SIMD vector (imm6 != 000xxx && Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>, #<fbits>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if op<1> == '0' && !HaveFP16Ext() then UNDEFINED;
if op<1> == '0' && imm6 IN {'10xxxx'} then UNDEFINED;
if imm6 IN {'0xxxxx'} then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
to_fixed = (op<0> == '1'); frac_bits = 64 - UInt(imm6);
unsigned = (U == '1');
integer esize;
integer elements;
case op<1> of
    when '0' esize = 16; elements = 4;
    when '1' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem\[D\[m+r\],e,esize\];
            if to_fixed then
                result = FPToFixed(op1, frac_bits, unsigned, StandardFPSCRValue\(\),
                    FPRounding\_ZERO);
            else
                result = FixedToFP(op1, frac_bits, unsigned, StandardFPSCRValue\(\),
                    FPRounding\_TIEEVEN);
            Elem\[D\[d+r\],e,esize\] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [FPFXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	1	op	1	U	Vd				1	0	sf	sx	1	i	0	imm4				
cond																															

Half-precision scalar (op == 0 && sf == 01)

(FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)

(FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && cond != '1110' then UNPREDICTABLE;
to_fixed = (op == '1'); unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
integer fp_size;
integer d;
case sf of
  when '01' fp_size = 16; d = UInt(Vd:D);
  when '10' fp_size = 32; d = UInt(Vd:D);
  when '11' fp_size = 64; d = UInt(D:Vd);
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				

Half-precision scalar (op == 0 && sf == 01) (FEAT_FP16)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01) (FEAT_FP16)

```
VCVT{<c>}{<q>}.<dt>.F16 <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 0 && sf == 10)

```
VCVT{<c>}{<q>}.F32.<dt> <Sdm>, <Sdm>, #<fbits>
```

Single-precision scalar (op == 1 && sf == 10)

```
VCVT{<c>}{<q>}.<dt>.F32 <Sdm>, <Sdm>, #<fbits>
```

Double-precision scalar (op == 0 && sf == 11)

```
VCVT{<c>}{<q>}.F64.<dt> <Ddm>, <Ddm>, #<fbits>
```

Double-precision scalar (op == 1 && sf == 11)

```
VCVT{<c>}{<q>}.<dt>.F64 <Ddm>, <Ddm>, #<fbits>
```

```
if sf == '00' || (sf == '01' && !HaveFP16Ext()) then UNDEFINED;
if sf == '01' && InITBlock() then UNPREDICTABLE;
to_fixed = (op == '1'); unsigned = (U == '1');
size = if sx == '0' then 16 else 32;
frac_bits = size - UInt(imm4:i);
integer fp_size;
integer d;
case sf of
  when '01' fp_size = 16; d = UInt(Vd:D);
  when '10' fp_size = 32; d = UInt(Vd:D);
  when '11' fp_size = 64; d = UInt(D:Vd);
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);
  if to_fixed then
    bits(size) result;
    case fp_size of
      when 16
        result = FPToFixed(S[d]<15:0>, frac_bits, unsigned, FPSCR[], FPRounding\_ZERO);
        S[d] = Extend(result, 32, unsigned);
      when 32
        result = FPToFixed(S[d], frac_bits, unsigned, FPSCR[], FPRounding\_ZERO);
        S[d] = Extend(result, 32, unsigned);
      when 64
        result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[], FPRounding\_ZERO);
        D[d] = Extend(result, 64, unsigned);
    else
      case fp_size of
        when 16
          bits(16) fp16 = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding\_TIEEVEN);
          S[d] = Zeros(16):fp16;
        when 32
          S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding\_TIEEVEN);
        when 64
          D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[], FPRounding\_TIEEVEN);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVT (between floating-point and integer, Advanced SIMD)

Vector Convert between floating-point and integer converts each element in a vector from floating-point to integer, or from integer to floating-point, and places the results in a second vector.

The vector elements are the same type, and are floating-point numbers or integers. Signed and unsigned integers are distinct.

The floating-point to integer operation uses the Round towards Zero rounding mode. The integer to floating-point operation uses the Round to Nearest rounding mode.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1	Vd	0	1	1	op	Q	M	0	Vm								

64-bit SIMD vector (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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								

64-bit SIMD vector (Q == 0)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVT{<c>}{<q>}.<dt1>.<dt2> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (op<1> == '1'); unsigned = (op<0> == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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();
    bits(esize) result;
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[m+r],e,esize];
            if to_integer then
                result = FPToFixed(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_ZERO);
            else
                result = FixedToFP(op1, 0, unsigned, StandardFPSCRValue(), FPRounding_TIEEVEN);
            Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

Half-precision to single-precision (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)

Single-precision to half-precision (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16; elements = 4;
m = UInt(M:Vm); 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							

Half-precision to single-precision (op == 1)

VCVT{<c>}{<q>}.F32.F16 <Qd>, <Dm> // (Encoded as op = 1)

Single-precision to half-precision (op == 0)

VCVT{<c>}{<q>}.F16.F32 <Dd>, <Qm> // (Encoded as op = 0)

```
if size != '01' then UNDEFINED;
half_to_single = (op == '1');
if half_to_single && Vd<0> == '1' then UNDEFINED;
if !half_to_single && Vm<0> == '1' then UNDEFINED;
esize = 16; elements = 4;
m = UInt(M:Vm); d = UInt(D:Vd);
```

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](#).
- <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\(\);
  for e = 0 to elements-1
    if half_to_single then
      Elem\[Q\[d>>1\],e,32\] = FPConvert\(Elem\[Din\[m\],e,16\], StandardFPSCRValue\\(\\)\);
    else
      Elem\[D\[d\],e,16\] = FPConvert\(Elem\[Qin\[m>>1\],e,32\], StandardFPSCRValue\\(\\)\);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																	

**Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)**

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

**Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)**

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

```
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

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																

**Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)**

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

**Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)**

VCVT{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVT{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVT{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVT{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVT{<c>}{<q>}.S32.F64 <Sd>, <Dm>

```

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);

```

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](#).
- <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);
  if to_integer then
    case esize of
      when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
      when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
      when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
    else
      case esize of
        when 16
          bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
          S[d] = Zeros(16):fp16;
        when 32
          S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
        when 64
          D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
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			

T1

VCVT{<c>}{<q>}.BF16.F32 <Dd>, <Qm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
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\(\);

    operand = Q\[m>>1\];
    for e = 0 to 3
        bits(32) op = Elem[operand, e, 32];
        Elem[result, e, 16] = FPConvertBF(op, StandardFPSCRValue\(\));
    D\[d\] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

Half-precision scalar (size == 01)

(FEAT_FP16)

VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>

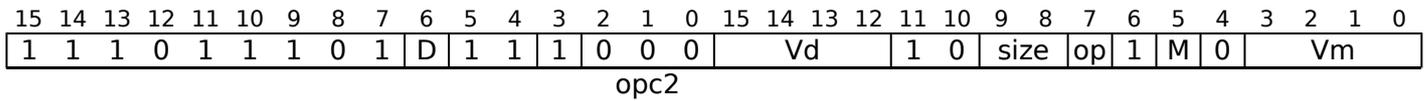
```
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

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



Half-precision scalar (size == 01) (FEAT_FP16)

VCVT{<c>}{<q>}.F16.<dt> <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVT{<c>}{<q>}.F32.<dt> <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVT{<c>}{<q>}.F64.<dt> <Dd>, <Sm>

```

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);

```

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(); CheckVFPEnabled(TRUE);
    if to_integer then
        case esize of
            when 16
                S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
        else
            case esize of
                when 16
                    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                when 64
                    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
            end
        end
    end

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

`VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>`

128-bit SIMD vector (Q == 1)

`VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>`

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

`VCVTA{<q>}.<dt>.<dt2> <Dd>, <Dm>`

128-bit SIMD vector (Q == 1)

`VCVTA{<q>}.<dt>.<dt2> <Qd>, <Qm>`

```

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
bits(esize) result;  
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,  
    StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTA (floating-point)

Convert floating-point to integer with Round to Nearest with Ties to Away converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest with Ties to Away rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	0		Vd			1	0	!= 00	op	1	M	0			Vm		
														RM				size													

Half-precision scalar (size == 01) (FEAT_FP16)

```
VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; 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	0	1	D	1	1	1	1	0	0		Vd			1	0	!= 00	op	1	M	0			Vm		
														RM				size													

Half-precision scalar (size == 01) (FEAT_FP16)

VCVTA{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTA{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTA{<q>}.<dt>.F64 <Sd>, <Dm>

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
```

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(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTB

Convert to or from a half-precision value in the bottom half of a single-precision register does one of the following:

- Converts the half-precision value in the bottom half of a single-precision register to single-precision and writes the result to a single-precision register.
- Converts the half-precision value in the bottom half of a single-precision register to double-precision and writes the result to a double-precision register.
- Converts the single-precision value in a single-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.
- Converts the double-precision value in a double-precision register to half-precision and writes the result into the bottom half of a single-precision register, preserving the other half of the destination register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	0	1	op	Vd				1	0	1	sz	0	1	M	0	Vm			
cond																T															

Half-precision to single-precision (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision (op == 1 && sz == 0)

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision (op == 1 && sz == 1)

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

```

uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = 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															

Half-precision to single-precision (op == 0 && sz == 0)

VCVTB{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision (op == 0 && sz == 1)

VCVTB{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision (op == 1 && sz == 0)

VCVTB{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision (op == 1 && sz == 1)

VCVTB{<c>}{<q>}.F16.F64 <Sd>, <Dm>

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); 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.
- <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(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        else
            S[d] = FPConvert(hp, FPSCR[]);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        else
            hp = FPConvert(S[m], FPSCR[]);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTB (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result into the bottom half of a single precision register, preserving the top 16 bits of the destination register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	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																															

A1

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
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			

T1

VCVTB{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
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], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

64-bit SIMD vector (Q == 0)

VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VCVTM{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTM{<q>}.<dt>.<dt2> <Qd>, <Qm>

```

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
bits(esize) result;  
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,  
    StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the *CPACR*, *NSACR*, *HCPTTR*, and *FPEXC* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
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													

Half-precision scalar (size == 01) (FEAT_FP16)

```
VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; 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	0	1	D	1	1	1	1	1	1	Vd				1	0	!= 00	op	1	M	0				Vm	
														RM				size													

Half-precision scalar (size == 01) (FEAT_FP16)

VCVTM{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTM{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTM{<q>}.<dt>.F64 <Sd>, <Dm>

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
```

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(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTN (Advanced SIMD)

Vector Convert floating-point to integer with Round to Nearest converts each element in a vector from floating-point to integer using the Round to Nearest rounding mode, and places the results in a second vector.

The operand vector elements are floating-point numbers.

The result vector elements are integers, and the same size as the operand vector elements. Signed and unsigned integers are distinct.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	1	Vd	0	0	0	1	op	Q	M	0								Vm

RM

64-bit SIMD vector (Q == 0)

`VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>`

128-bit SIMD vector (Q == 1)

`VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>`

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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	1	D	1	1	size	1	1	Vd	0	0	0	1	op	Q	M	0							Vm

RM

64-bit SIMD vector (Q == 0)

`VCVTN{<q>}.<dt>.<dt2> <Dd>, <Dm>`

128-bit SIMD vector (Q == 1)

`VCVTN{<q>}.<dt>.<dt2> <Qd>, <Qm>`

```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
bits(esize) result;  
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,  
    StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTN (floating-point)

Convert floating-point to integer with Round to Nearest converts a value in a register from floating-point to a 32-bit integer using the Round to Nearest rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	0	1	Vd				1	0	!= 00	op	1	M	0	Vm				
RM														size																	

Half-precision scalar (size == 01) (FEAT_FP16)

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
    when '01' esize = 16; m = UInt(Vm:M);
    when '10' esize = 32; m = UInt(Vm:M);
    when '11' esize = 64; 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	0	1	D	1	1	1	1	0	1	Vd				1	0	!= 00	op	1	M	0	Vm				
RM														size																	

Half-precision scalar (size == 01) (FEAT_FP16)

VCVTN{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTN{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTN{<q>}.<dt>.F64 <Sd>, <Dm>

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
```

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(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

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

64-bit SIMD vector (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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	1	D	1	1	size	1	1	Vd					0	0	1	0	op	Q	M	0			Vm

RM

64-bit SIMD vector (Q == 0)

VCVTP{<q>}.<dt>.<dt2> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VCVTP{<q>}.<dt>.<dt2> <Qd>, <Qm>

```

if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
bits(esize) result;  
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,  
    StandardFPSCRValue(), rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTP (floating-point)

Convert floating-point to integer with Round towards +Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards +Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	0		Vd			1	0	!= 00	op	1	M	0			Vm		
														RM				size													

Half-precision scalar (size == 01) (FEAT_FP16)

```
VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; 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	0	1	D	1	1	1	1	1	0		Vd			1	0	!= 00	op	1	M	0			Vm		
														RM				size													

Half-precision scalar (size == 01) (FEAT_FP16)

VCVTP{<q>}.<dt>.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VCVTP{<q>}.<dt>.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VCVTP{<q>}.<dt>.F64 <Sd>, <Dm>

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); unsigned = (op == '0');
d = UInt(Vd:D);
integer esize;
integer m;
case size of
  when '01' esize = 16; m = UInt(Vm:M);
  when '10' esize = 32; m = UInt(Vm:M);
  when '11' esize = 64; m = UInt(M:Vm);
```

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(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																	

**Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)**

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

**Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)**

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>

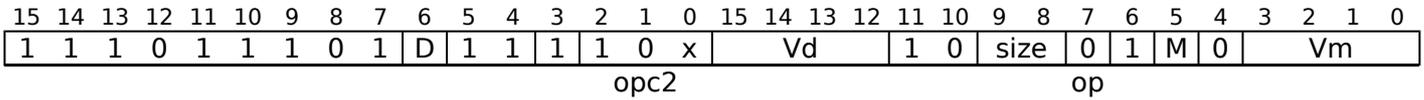
```
if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer d;
integer esize;
integer m;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);
```

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



**Half-precision scalar (opc2 == 100 && size == 01)
(FEAT_FP16)**

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

**Half-precision scalar (opc2 == 101 && size == 01)
(FEAT_FP16)**

VCVTR{<c>}{<q>}.S32.F16 <Sd>, <Sm>

Single-precision scalar (opc2 == 100 && size == 10)

VCVTR{<c>}{<q>}.U32.F32 <Sd>, <Sm>

Single-precision scalar (opc2 == 101 && size == 10)

VCVTR{<c>}{<q>}.S32.F32 <Sd>, <Sm>

Double-precision scalar (opc2 == 100 && size == 11)

VCVTR{<c>}{<q>}.U32.F64 <Sd>, <Dm>

Double-precision scalar (opc2 == 101 && size == 11)

VCVTR{<c>}{<q>}.S32.F64 <Sd>, <Dm>

```

if opc2 != '000' && !(opc2 IN {'10x'}) then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
integer esize;
integer m;
integer d;
boolean unsigned;
FPRounding rounding;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
    d = UInt(Vd:D);
    case size of
        when '01' esize = 16; m = UInt(Vm:M);
        when '10' esize = 32; m = UInt(Vm:M);
        when '11' esize = 64; m = UInt(M:Vm);
else
    unsigned = (op == '0');
    rounding = FPRoundingMode(FPSCR[]);
    m = UInt(Vm:M);
    case size of
        when '01' esize = 16; d = UInt(Vd:D);
        when '10' esize = 32; d = UInt(Vd:D);
        when '11' esize = 64; d = UInt(D:Vd);

```

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](#).
- <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);
  if to_integer then
    case esize of
      when 16
        S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
      when 32
        S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
      when 64
        S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
    else
      case esize of
        when 16
          bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
          S[d] = Zeros(16):fp16;
        when 32
          S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
        when 64
          D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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															

Half-precision to single-precision (op == 0 && sz == 0)

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision (op == 0 && sz == 1)

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision (op == 1 && sz == 0)

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision (op == 1 && sz == 1)

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

```

uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); m = 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															

Half-precision to single-precision (op == 0 && sz == 0)

VCVTT{<c>}{<q>}.F32.F16 <Sd>, <Sm>

Half-precision to double-precision (op == 0 && sz == 1)

VCVTT{<c>}{<q>}.F64.F16 <Dd>, <Sm>

Single-precision to half-precision (op == 1 && sz == 0)

VCVTT{<c>}{<q>}.F16.F32 <Sd>, <Sm>

Double-precision to half-precision (op == 1 && sz == 1)

VCVTT{<c>}{<q>}.F16.F64 <Sd>, <Dm>

```
uses_double = (sz == '1'); convert_from_half = (op == '0');
lowbit = (if T == '1' then 16 else 0);
integer d;
integer m;
if uses_double then
    if convert_from_half then
        d = UInt(D:Vd); m = UInt(Vm:M);
    else
        d = UInt(Vd:D); m = UInt(M:Vm);
else
    d = UInt(Vd:D); 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.
- <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(); CheckVFPEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        else
            S[d] = FPConvert(hp, FPSCR[]);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        else
            hp = FPConvert(S[m], FPSCR[]);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VCVTT (BFloat16)

Converts the single-precision value in a single-precision register to BFloat16 format and writes the result in the top half of a single-precision register, preserving the bottom 16 bits of the register.

Unlike the BFloat16 multiplication instructions, this instruction honors all the control bits in the *FPSCR* that apply to single-precision arithmetic, including the rounding mode. This instruction can generate a floating-point exception which causes a cumulative exception bit in the *FPSCR* to be set, or a synchronous exception to be taken, depending on the enable bits in the *FPSCR*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	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																															

A1

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
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			

T1

VCVTT{<c>}{<q>}.BF16.F32 <Sd>, <Sm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
integer d = UInt(Vd:D);
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();
    CheckVFPEnabled(TRUE);

    S[d]<31:16> = FPConvertBF(S[m], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																															

Half-precision scalar (size == 01) (FEAT_FP16)

VDIV{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VDIV{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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				

Half-precision scalar (size == 01) (FEAT_FP16)

VDIV{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VDIV{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VDIV{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zeros(16) : FPDiv(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
    when 32
      S[d] = FPDiv(S[n], S[m], FPSCR[]);
    when 64
      D[d] = FPDiv(D[n], D[m], FPSCR[]);
```

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			

64-bit SIMD vector (Q == 0)

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M);
integer i = UInt(M);
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			

64-bit SIMD vector (Q == 0)

```
VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VDOT{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M);
integer i = UInt(M);
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;
bits(64) operand2;
bits(64) result;

CheckAdvSIMDEnabled();

operand2 = Din[m];
for r = 0 to regs-1
  operand1 = Din[n+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * i + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * i + 1, 16];
    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
  D[d+r] = result;

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDOT (vector)

BFloat16 floating-point (BF16) dot product (vector). This instruction delimits the source vectors into pairs of 16-bit BF16 elements. Within each pair, the elements in the first source vector are multiplied by the corresponding elements in the second source vector. The resulting single-precision products are then summed and added destructively to the single-precision element in the destination vector which aligns with the pair of BF16 values in the first source vector. The instruction does not update the *FPSCR* exception status.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32BF16)

31	30	29	28	27	26	25	24	23	22	21	20	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			

64-bit SIMD vector (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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			

64-bit SIMD vector (Q == 0)

VDOT{<q>}.BF16 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VDOT{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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

```
bits(64) operand1;
bits(64) operand2;
bits(64) result;
```

```
CheckAdvSIMDEnabled();
```

```
for r = 0 to regs-1
  operand1 = Din[n+r];
  operand2 = Din[m+r];
  result = Din[d+r];
  for e = 0 to 1
    bits(16) elt1_a = Elem[operand1, 2 * e + 0, 16];
    bits(16) elt1_b = Elem[operand1, 2 * e + 1, 16];
    bits(16) elt2_a = Elem[operand2, 2 * e + 0, 16];
    bits(16) elt2_b = Elem[operand2, 2 * e + 1, 16];
    bits(32) sum = BFAdd(BFMul(elt1_a, elt2_a), BFMul(elt1_b, elt2_b));
    Elem[result, e, 32] = BFAdd(Elem[result, e, 32], sum);
  D[d+r] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VDUP (general-purpose register)

Duplicate general-purpose register to vector duplicates an element from a general-purpose register into every element of the destination vector.

The destination vector elements can be 8-bit, 16-bit, or 32-bit fields. The source element is the least significant 8, 16, or 32 bits of the general-purpose register. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [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	1	0	1	B	Q	0	Vd						Rt				1	0	1	1	D	0	E	1	(0)	(0)	(0)	(0)
cond																															

A1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)
```

```
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
case B:E of
  when '00' esize = 32; elements = 2;
  when '01' esize = 16; elements = 4;
  when '10' esize = 8; elements = 8;
  when '11' UNDEFINED;
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
1	1	1	0	1	1	1	0	1	B	Q	0	Vd						Rt				1	0	1	1	D	0	E	1	(0)	(0)	(0)	(0)

T1

```
VDUP{<c>}{<q>}.<size> <Qd>, <Rt> // (Encoded as Q = 1)
```

```
VDUP{<c>}{<q>}.<size> <Dd>, <Rt> // (Encoded as Q = 0)
```

```
if Q == '1' && Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
case B:E of
  when '00' esize = 32; elements = 2;
  when '01' esize = 16; elements = 4;
  when '10' esize = 8; elements = 8;
  when '11' UNDEFINED;
if t == 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](#). Arm strongly recommends that any VDUP instruction is unconditional, see [Conditional execution](#).
- <q> See [Standard assembler syntax fields](#).
- <size> The data size for the elements of the destination vector. It must be one of:
- 8**
Encoded as [b, e] = 0b10.
 - 16**
Encoded as [b, e] = 0b01.
 - 32**
Encoded as [b, e] = 0b00.
- <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\(\);
    scalar = R[t]<size-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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

(Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer index;
case imm4 of
  when 'xxx1' esize = 8; elements = 8; index = UInt(imm4<3:1>);
  when 'xx10' esize = 16; elements = 4; index = UInt(imm4<3:2>);
  when 'x100' esize = 32; elements = 2; index = UInt(imm4<3>);
d = UInt(D:Vd); m = UInt(M:Vm); 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			

(Q == 0)

```
VDUP{<c>}{<q>}.<size> <Dd>, <Dm[x]>
```

(Q == 1)

```
VDUP{<c>}{<q>}.<size> <Qd>, <Dm[x]>
```

```
if imm4 IN {'x000'} then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer index;
case imm4 of
  when 'xxx1' esize = 8; elements = 8; index = UInt(imm4<3:1>);
  when 'xx10' esize = 16; elements = 4; index = UInt(imm4<3:2>);
  when 'x100' esize = 32; elements = 2; index = UInt(imm4<3>);
d = UInt(D:Vd); m = UInt(M:Vm); 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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VEOR

Vector Bitwise Exclusive OR performs a bitwise Exclusive OR operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	0	0	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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			

64-bit SIMD vector (Q == 0)

```
VEOR{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VEOR{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

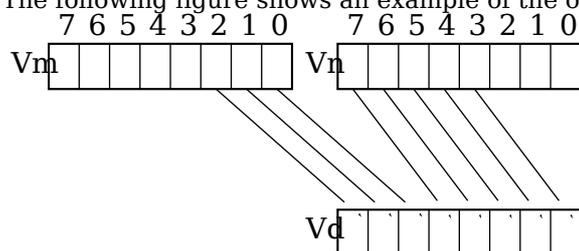
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.8 {<Dd>}, <Dn>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>}, <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1'); position = 8 * UInt(imm4);
d = UInt(D:Vd); n = UInt(N:Vn); 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			

64-bit SIMD vector (Q == 0)

```
VEXT{<c>}{<q>}.8 {<Dd>}, <Dn>, <Dm>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VEXT{<c>}{<q>}.8 {<Qd>}, <Qn>, <Qm>, #<imm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if Q == '0' && imm4<3> == '1' then UNDEFINED;
quadword_operation = (Q == '1'); position = 8 * UInt(imm4);
d = UInt(D:Vd); n = UInt(N:Vn); 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](#).

<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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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			

64-bit SIMD vector (Q == 0)

VEXT{<c>}{<q>}.<size> {<Dd>}, <Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>}, <Dn>, <Dm>, #<imm*(size/8)>

128-bit SIMD vector (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			

64-bit SIMD vector (Q == 0)

VEXT{<c>}{<q>}.<size> {<Dd>}, <Dn>, <Dm>, #<imm>

is equivalent to

VEXT{<c>}{<q>}.8 {<Dd>}, <Dn>, <Dm>, #<imm*(size/8)>

128-bit SIMD vector (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 [Standard assembler syntax fields](#). This encoding must be unconditional.
For encoding T1: see [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

<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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMA

Vector Fused Multiply Accumulate multiplies corresponding elements of two vectors, and accumulates the results into the elements of the destination vector. The instruction does not round the result of the multiply before the accumulation.

Depending on settings in the [CPACR](#), [NSACR](#), [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

64-bit SIMD vector (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
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

Half-precision scalar (size == 01)
(FEAT_FP16)

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

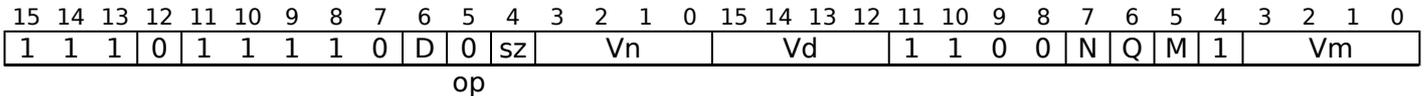
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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



64-bit SIMD vector (Q == 0)

VFMA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

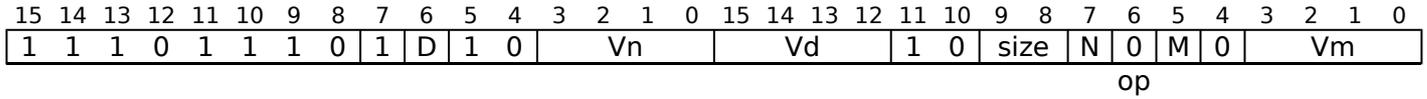
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
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



Half-precision scalar (size == 01) (FEAT_FP16)

VFMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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
        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);
                Elem[D[d+r],e,esize] = FPMuLAdd(Elem[D[d+r],e,esize],
                    op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMuLAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMuLAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMuLAdd(D[d], op64, D[m], FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
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			

T1

```
VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<2:0>);
integer i = UInt(M:Vm<3>);
integer elements = 128 DIV 32;
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.
- <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();
bits(128) operand1 = Q[n>>1];
bits(64) operand2 = D[m];
bits(128) operand3 = Q[d>>1];
bits(128) result;

bits(32) element2 = Elem[operand2, i, 16] : Zeros(16);

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                  StandardFPSCRValue());
Q[d>>1] = result;

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 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			

A1

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
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			

T1

VFMA<bt>{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer elements = 128 DIV 32;
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();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) operand3 = Q[d>>1];
bits(128) result;

for e = 0 to elements-1
    bits(32) element1 = Elem[operand1, 2 * e + sel, 16] : Zeros(16);
    bits(32) element2 = Elem[operand2, 2 * e + sel, 16] : Zeros(16);
    bits(32) addend = Elem[operand3, e, 32];
    Elem[result, e, 32] = FPMulAdd(addend, element1, element2,
                                  StandardFPSCRValue());

Q[d>>1] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

64-bit SIMD vector (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

128-bit SIMD vector (Q == 1)

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

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																															

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='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.
<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;

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);
        Elem[result, e, esize] = FPMuLAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal);
    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)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
																	S														

64-bit SIMD vector (Q == 0)

VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector (Q == 1)

VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
```

```
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

T1

(FEAT_FHM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0	Vn			Vd			1	0	0	0	N	Q	M	1	Vm					
																	S														

64-bit SIMD vector (Q == 0)

```
VFMAL{<q>}.F16 <Dd>, <Sn>, <Sm>
```

128-bit SIMD vector (Q == 1)

```
VFMAL{<q>}.F16 <Qd>, <Dn>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='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.
<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();
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);
        Elem[result, e, esize] = FPMu1AddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal);
    D[d+r] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VFMS

Vector Fused Multiply Subtract negates the elements of one vector and multiplies them with the corresponding elements of another vector, adds the products to the corresponding elements of the destination vector, and places the results in the destination vector. The instruction does not round the result of the multiply before the addition.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	sz	Vn				Vd				1	1	0	0	N	Q	M	1	Vm			

op

64-bit SIMD vector (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
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	1	M	0	Vm				

cond op

Half-precision scalar (size == 01)
(FEAT_FP16)

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

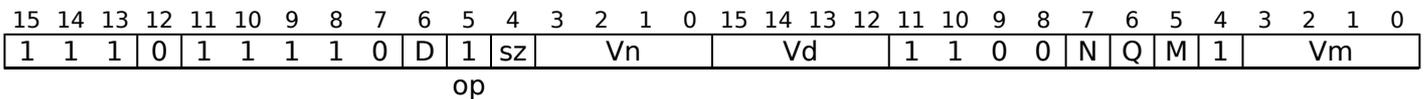
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE; op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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



64-bit SIMD vector (Q == 0)

VFMS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VFMS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; op1_neg = (op == '1');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
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	1	D	1	0	Vn			Vd			1	0	size	N	1	M	0	Vm						
																op															

Half-precision scalar (size == 01) (FEAT_FP16)

VFMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE; opl_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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
        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);
                Elem[D[d+r],e,esize] = FPMuLAdd(Elem[D[d+r],e,esize],
                    op1, Elem[D[m+r],e,esize], StandardFPSCRValue());

    else // VFP instruction
        case esize of
            when 16
                op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
                S[d] = Zeros(16) : FPMuLAdd(S[d]<15:0>, op16, S[m]<15:0>, FPSCR[]);
            when 32
                op32 = if op1_neg then FPNeg(S[n]) else S[n];
                S[d] = FPMuLAdd(S[d], op32, S[m], FPSCR[]);
            when 64
                op64 = if op1_neg then FPNeg(D[n]) else D[n];
                D[d] = FPMuLAdd(D[d], op64, D[m], FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

64-bit SIMD vector (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]

128-bit SIMD vector (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
```

```
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);
```

```
integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

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																				

64-bit SIMD vector (Q == 0)

```
VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(Vm<2:0>) else UInt(Vm<2:0>:M);

integer index = if Q == '1' then UInt(M:Vm<3>) else UInt(Vm<3>);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='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.
<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;

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);
        Elem[result, e, esize] = FPMuLAddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal);
    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																															

64-bit SIMD vector (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>

```
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;
```

```
integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='1';
```

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																															

64-bit SIMD vector (Q == 0)

VFMSL{<q>}.F16 <Dd>, <Sn>, <Sm>

128-bit SIMD vector (Q == 1)

VFMSL{<q>}.F16 <Qd>, <Dn>, <Dm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16MulNoRoundingToFP32Ext() then UNDEFINED;
if Q == '1' && Vd<0> == '1' then UNDEFINED;

integer d = UInt(D:Vd);
integer n = if Q == '1' then UInt(N:Vn) else UInt(Vn:N);
integer m = if Q == '1' then UInt(M:Vm) else UInt(Vm:M);
integer esize = 32;
integer regs = if Q=='1' then 2 else 1;
integer datasize = if Q=='1' then 64 else 32;
boolean sub_op = S=='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.
- <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();
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);
        Elem[result, e, esize] = FPMu1AddH(Elem[operand3, e, esize], element1, element2, StandardFPSCRVal);
    D[d+r] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

Half-precision scalar (size == 01) (FEAT_FP16)

VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
opl_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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															

Half-precision scalar (size == 01) (FEAT_FP16)

VFNMA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFNMA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFNMA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16
      op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
      S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
    when 32
      op32 = if op1_neg then FPNeg(S[n]) else S[n];
      S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
    when 64
      op64 = if op1_neg then FPNeg(D[n]) else D[n];
      D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
```


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																			

Half-precision scalar (size == 01) (FEAT_FP16)

VFNMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VFNMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VFNMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
opl_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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																			

Half-precision scalar (size == 01) (FEAT_FP16)

```
VFNMMS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VFNMMS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VFNMMS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
op1_neg = (op == '1');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  case esize of
    when 16
      op16 = if op1_neg then FPNeg(S[n]<15:0>) else S[n]<15:0>;
      S[d] = Zeros(16) : FPMulAdd(FPNeg(S[d]<15:0>), op16, S[m]<15:0>, FPSCR[]);
    when 32
      op32 = if op1_neg then FPNeg(S[n]) else S[n];
      S[d] = FPMulAdd(FPNeg(S[d]), op32, S[m], FPSCR[]);
    when 64
      op64 = if op1_neg then FPNeg(D[n]) else D[n];
      D[d] = FPMulAdd(FPNeg(D[d]), op64, D[m], FPSCR[]);
```


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									

64-bit SIMD vector (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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									

64-bit SIMD vector (Q == 0)

VHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:l>;
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn	Vd	0	0	1	0	N	Q	M	0	Vm	op									

64-bit SIMD vector (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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									

64-bit SIMD vector (Q == 0)

VHSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VHSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
add = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            result = if add then op1+op2 else op1-op2;
            Elem[D[d+r],e,esize] = result<esize:l>;
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

VINS{<q>}.F16 <Sd>, <Sm>

```
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); 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			

T1

VINS{<q>}.F16 <Sd>, <Sm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); 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(); CheckVFPEnabled(TRUE);
    S[d] = S[m]<15:0> : S[d]<15:0>;
```


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																															

A1

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

```
if !HaveFJCVTZSExt() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
d = UInt(Vd:D); 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			

T1

VJCVT{<q>}.S32.F64 <Sd>, <Dm>

```
if !HaveFJCVTZSExt() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
d = UInt(Vd:D); 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();
CheckVFPEEnabled(TRUE);
bits(64) fltval = D[m];
bits(32) intval;
bit Z;
(intval, Z) = FPToFixedJS(fltval, FPSCR[], FALSE);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
```


VLD1 (multiple single elements)

Load multiple single 1-element structures to one, two, three, or four registers loads elements from memory into one, two, three, or four registers, without de-interleaving. Every element of each register is loaded. For details of the addressing mode, see *Advanced SIMD addressing mode*.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#) , [A2](#) , [A3](#) and [A4](#)) and T32 ([T1](#) , [T2](#) , [T3](#) and [T4](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	0	D	1	0				Rn				Vd		0	1	1	1	size	align			Rm		

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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		

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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			

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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			

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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							

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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							

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    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
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                data<31:0> = if BigEndian(AccType_NORMAL) then MemU[address+4,4] else MemU[address,4];
                data<63:32> = if BigEndian(AccType_NORMAL) then MemU[address,4] else MemU[address+4,4];
                Elem[D[d+r],e] = 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size); regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' || (size == '00' && a == '1') then UNDEFINED;
ebytes = 1 << UInt(size); regs = if T == '0' then 1 else 2;
alignment = if a == '0' then 1 else ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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();
  address = R[n]; iswrite = FALSE;
  - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
  bits(64) replicated_element = Replicate(MemU[address,ebytes]);
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [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	0	index_align		Rm					
size																															

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																															

Offset (Rm == 1111)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
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;
ebytes = 4; index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
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	1	1	0	0	1	1	D	1	0		Rn		Vd					0	0	0	0	index_align				Rm			

size

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	1	D	1	0		Rn		Vd					0	1	0	0	index_align				Rm			

size

Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

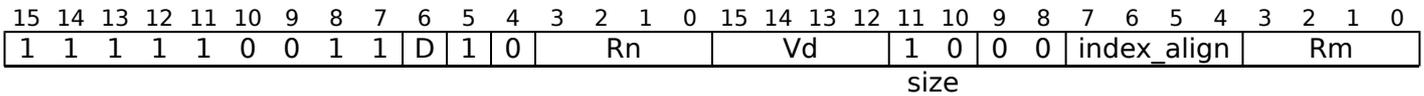
VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD1 (single element to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

```

T3



Offset (Rm == 1111)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

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;
ebytes = 4; index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d],index] = 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.

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																															

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```

pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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					

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

pairs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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							

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            Elem[D[d+r], e] = MemU[address, ebytes];
            Elem[D[d2+r], e] = 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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		

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

```
if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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		

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
ebytes = 1 << UInt(size);
alignment = if a == '0' then 1 else 2*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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 [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 two SIMD&FP registers.
The list must be one of:

{ <Dd>[], <Dd+1>[] }

Single-spaced registers, encoded in the "T" field as 0.

{ <Dd>[], <Dd+2>[] }

Double-spaced registers, encoded in the "T" field as 1.

The register <Dd> is encoded in the "D:Vd" field.

<Rn> Is the general-purpose base register, encoded in the "Rn" field.

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the "a" field as 0.

Whenever <align> is present, the permitted values and encoding depend on <size>:

<size> == 8

<align> is 16, meaning 16-bit alignment, encoded in the "a" field as 1.

<size> == 16

<align> is 32, meaning 32-bit alignment, encoded in the "a" field as 1.

<size> == 32

<align> is 64, meaning 64-bit alignment, encoded in the "a" field as 1.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about the variants of this instruction, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled\(\);
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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	1	index_align			Rm				
size																															

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	1	0	1	0	0	1	D	1	0	Rn				Vd				0	1	0	1	index_align			Rm				
size																															

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

31	30	29	28	27	26	25	24	23	22	21	20	19	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	1			index_align			Rm	
size																															

Offset (Rm == 1111)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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			0	0	0	1	index_align			Rm						
																size															

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	1	D	1	0	Rn			Vd			0	1	0	1	index_align			Rm						
																size															

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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	1	index_align				Rm			
																size															

Offset (Rm == 1111)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD2 (single 2-element structure to all lanes)";
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    Elem[D[d], index] = MemU[address, ebytes];
    Elem[D[d2], index] = MemU[address+ebytes, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 *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	1	0	0	0	D	1	0				Rn				Vd		0	1	0	x	size	align			Rm		

itype

Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
integer inc;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
integer inc;
case itype of
  when '0100'
    inc = 1;
  when '0101'
    inc = 2;
  otherwise
    SEE "Related encodings";
if size == '11' || align<1> == '1' then UNDEFINED;
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address,ebytes];
        Elem[D[d2],e] = MemU[address+ebytes,ebytes];
        Elem[D[d3],e] = MemU[address+2*ebytes,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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 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 *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	1	0	0	1	D	1	0			Rn			Vd			1	1	1	0	size	T	0			Rm		

a

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```
if size == '11' || a == '1' then UNDEFINED;
ebytes = 1 << UInt(size);
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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\(\);
    address = R\[n\];
    D\[d\] = Replicate\(MemU\[address,ebytes\]\);
    D\[d2\] = Replicate\(MemU\[address+ebytes,ebytes\]\);
    D\[d3\] = Replicate\(MemU\[address+2\*ebytes,ebytes\]\);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 *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	1	0		index_align			Rm	

size

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0				Rn				Vd			1	0	1	0		index_align			Rm	
																				size											

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				0	0	1	0	index_align				Rm			
																size															

Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]
```

Post-indexed (Rm == 1101)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!
```

Post-indexed (Rm != 11x1)

```
VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>
```

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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	0	index_align			Rm				
																size															

Offset (Rm == 1111)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VLD3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```
if size == '11' then SEE "VLD3 (single 3-element structure to all lanes)";
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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(); CheckAdvSIMDEnabled();
    address = R[n];
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
integer inc;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																															

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

integer inc;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    SEE "Related encodings";
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        Elem[D[d], e] = MemU[address, ebytes];
        Elem[D[d2], e] = MemU[address+ebytes, ebytes];
        Elem[D[d3], e] = MemU[address+2*ebytes, ebytes];
        Elem[D[d4], e] = MemU[address+3*ebytes, 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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	1	0	0	1	D	1	0			Rn				Vd			1	1	1	1	size	T	a			Rm	

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}],<Rm>
```

```
if size == '11' && a == '0' then UNDEFINED;
integer ebytes;
integer alignment;
if size == '11' then
    ebytes = 4; alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' && a == '0' then UNDEFINED;
integer ebytes;
integer alignment;
if size == '11' then
    ebytes = 4; alignment = 16;
else
    ebytes = 1 << UInt(size);
    if size == '10' then
        alignment = if a == '0' then 1 else 8;
    else
        alignment = if a == '0' then 1 else 4*ebytes;
inc = if T == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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 [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
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();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    D[d] = Replicate(MemU[address,ebytes]);
    D[d2] = Replicate(MemU[address+ebytes,ebytes]);
    D[d3] = Replicate(MemU[address+2*ebytes,ebytes]);
    D[d4] = Replicate(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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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	1	1		index_align			Rm	

size

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	1	0			Rn				Vd			1	0	1	1			index_align			Rm	
size																															

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				0	0	1	1	index_align		Rm					
size																															

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																															

Offset (Rm == 1111)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VLD4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then SEE "VLD4 (single 4-element structure to all lanes)";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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 [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();
    address = R[n]; iswrite = FALSE;
    - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
    Elem[D[d], index] = MemU[address,ebytes];
    Elem[D[d2],index] = MemU[address+ebytes,ebytes];
    Elem[D[d3],index] = MemU[address+2*ebytes,ebytes];
    Elem[D[d4],index] = 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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>															

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>
```

```
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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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																															

Decrement Before (P == 1 && U == 0 && W == 1)

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

Increment After (P == 0 && U == 1)

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

```

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 !)
single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); 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>															

Decrement Before (P == 1 && U == 0 && W == 1)

VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

```

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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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							

Decrement Before (P == 1 && U == 0 && W == 1)

```
VLDMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After (P == 0 && U == 1)

```
VLDM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
VLDMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
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 !)
single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); 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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used.

! 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
VPOP	P == '0' && U == '1' && W == '1' && Rn == '1101'

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
            word1 = MemA[address,4]; 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(AccType_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (immediate)

Load SIMD&FP register (immediate) loads a single register from the Advanced SIMD and floating-point register file, using an address from a general-purpose register, with an optional offset.

Depending on settings in the [CPACR](#), [NSACR](#), [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																				

Half-precision scalar (size == 01) (FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}{.32} <Sd>, [<Rn> {, #{+/-}<imm>}]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}{.64} <Dd>, [<Rn> {, #{+/-}<imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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																															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, [<Rn> {, #<+/-><imm>}]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}.64 <Dd>, [<Rn> {, #<+/-><imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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 *Standard assembler syntax fields*.

<q> See *Standard assembler syntax fields*.

.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":

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(); CheckVFPEnabled(TRUE);
  base = if n == 15 then Align(PC,4) else R[n];
  address = if add then (base + imm32) else (base - imm32);
  case esize of
    when 16
      S[d] = Zeros(16) : MemA[address,2];
    when 32
      S[d] = MemA[address,4];
    when 64
      word1 = MemA[address,4]; word2 = MemA[address+4,4];
      // Combine the word-aligned words in the correct order for current endianness.
      D[d] = if BigEndian(AccType\_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VLDR (literal)

Load SIMD&FP register (literal) loads a single register from the Advanced SIMD and floating-point register file, using an address from the PC value and an immediate offset.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information, see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	1	U	D	0	1	1	1	1	1	Vd				1	0	size	imm8								
cond				Rn																											

Half-precision scalar (size == 01) (FEAT_FP16)

```
VLDR{<c>}{<q>}.16 <Sd>, <label>
```

```
VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]
```

Single-precision scalar (size == 10)

```
VLDR{<c>}{<q>}.32 <Sd>, <label>
```

```
VLDR{<c>}{<q>}.32 <Sd>, [PC, #{+/-}<imm>]
```

Double-precision scalar (size == 11)

```
VLDR{<c>}{<q>}.64 <Dd>, <label>
```

```
VLDR{<c>}{<q>}.64 <Dd>, [PC, #{+/-}<imm>]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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	1	1	1	1	Vd				1	0	size	imm8								
Rn																															

Half-precision scalar (size == 01) (FEAT_FP16)

VLDR{<c>}{<q>}.16 <Sd>, <label>

VLDR{<c>}{<q>}.16 <Sd>, [PC, #{+/-}<imm>]

Single-precision scalar (size == 10)

VLDR{<c>}{<q>}.32 <Sd>, <label>

VLDR{<c>}{<q>}.32 <Sd>, [PC, #{+/-}<imm>]

Double-precision scalar (size == 11)

VLDR{<c>}{<q>}.64 <Dd>, <label>

VLDR{<c>}{<q>}.64 <Dd>, [PC, #{+/-}<imm>]

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

.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(); CheckVFPEnabled(TRUE);
    base = if n == 15 then Align(PC,4) else R[n];
    address = if add then (base + imm32) else (base - imm32);
    case esize of
        when 16
            S[d] = Zeros(16) : MemA[address,2];
        when 32
            S[d] = MemA[address,4];
        when 64
            word1 = MemA[address,4]; word2 = MemA[address+4,4];
            // Combine the word-aligned words in the correct order for current endianness.
            D[d] = if BigEndian(AccType_ATOMIC) 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMAX (floating-point)

Vector Maximum compares corresponding elements in two vectors, and copies the larger of each pair into the corresponding element in the destination vector.

The operand vector elements are floating-point numbers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	0	sz	Vn			Vd			1	1	1	1	N	Q	M	0	Vm					

op

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	0	Vm					

op

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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															

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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															

64-bit SIMD vector (Q == 0)

VMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAX{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Single-precision scalar (size == 10)

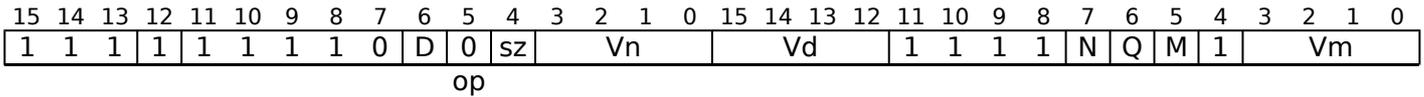
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Double-precision scalar (size == 11)

VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



64-bit SIMD vector (Q == 0)

VMAXNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMAXNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

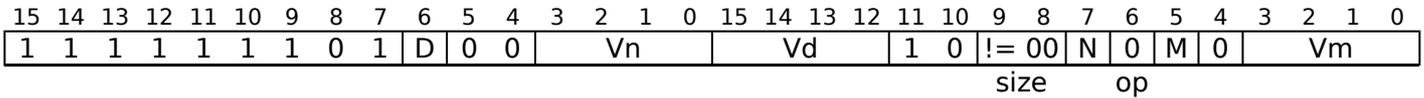
```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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



Half-precision scalar (size == 01)
(FEAT_FP16)

VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Single-precision scalar (size == 10)

VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Double-precision scalar (size == 11)

VMAXNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

```

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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();
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r],e,esize]; op2 = Elem[D[m+r],e,esize];
            if maximum then
                Elem[D[d+r],e,esize] = FPMax(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r],e,esize] = FPMin(op1, op2, StandardFPSCRValue());
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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									

64-bit SIMD vector (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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									

64-bit SIMD vector (Q == 0)

VMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMIN{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Int(Elem[D[n+r],e,esize], unsigned);
            op2 = Int(Elem[D[m+r],e,esize], unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	0	D	1	sz	Vn				Vd				1	1	1	1	N	Q	M	1	Vm			

op

64-bit SIMD vector (Q == 0)

VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	1	M	0	Vm				

size op

Half-precision scalar (size == 01)
(FEAT_FP16)

VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Single-precision scalar (size == 10)

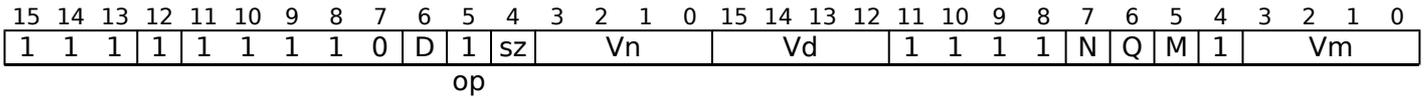
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Double-precision scalar (size == 11)

VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

T1



64-bit SIMD vector (Q == 0)

VMINNM{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMINNM{<q>}.<dt> <Qd>, <Qn>, <Qm>

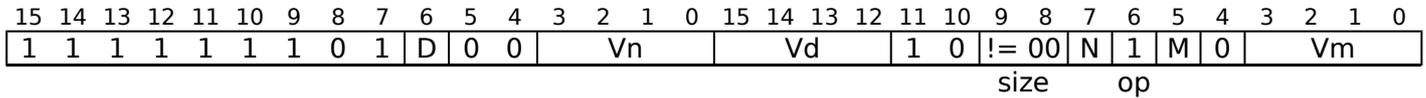
```
if InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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



Half-precision scalar (size == 01)
(FEAT_FP16)

VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Single-precision scalar (size == 10)

VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Double-precision scalar (size == 11)

VMINNM{<q>}.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

```

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
maximum = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[D[n+r], e, esize]; op2 = Elem[D[m+r], e, esize];
            if maximum then
                Elem[D[d+r], e, esize] = FPMaxNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMinNum(op1, op2, StandardFPSCRValue());
else // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaxNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            else
                S[d] = Zeros(16) : FPMinNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            if maximum then
                S[d] = FPMaxNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMinNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaxNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMinNum(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

64-bit SIMD vector (Q == 0)

```
VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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																				

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```

if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																			

Half-precision scalar (size == 01)
(FEAT_FP16)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

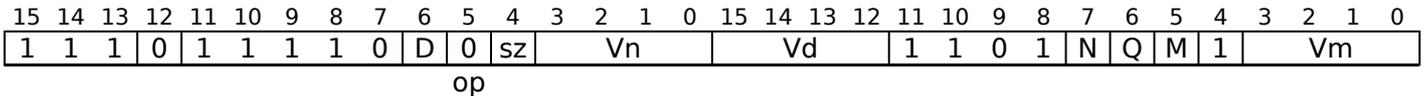
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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



64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	0	M	0	Vm						
																op															

Half-precision scalar (size == 01) (FEAT_FP16)

VMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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(); CheckAdvSIMDOrVFPEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>));
                S[d] = Zeros(16) : FPAAdd(S[d]<15:0>, addend16, FPSCR[]);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                S[d] = FPAAdd(S[d], addend32, FPSCR[]);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
                D[d] = FPAAdd(D[d], addend64, FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VMLA{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLA{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

A1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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																				

T1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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	0	0	N	0	M	0					Vm			
size											op																				

A1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	0	0	0	N	0	M	0					Vm			
size											op																				

T1

VMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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	1	0	F	N	1	M	0	Vm										
size											op																				

64-bit SIMD vector (Q == 0)

```
VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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	1	0	F	N	1	M	0	Vm										
size											op																				

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
add = (op == '0'); floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																			

Half-precision scalar (size == 01)
(FEAT_FP16)

VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

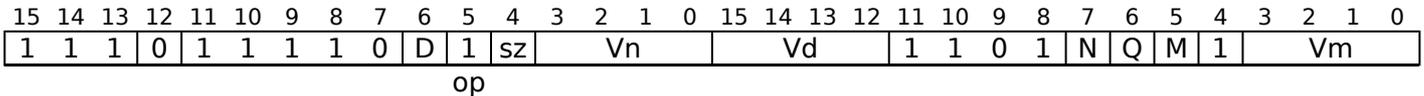
```

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



64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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															

Half-precision scalar (size == 01) (FEAT_FP16)

VMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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
        for r = 0 to regs-1
            for e = 0 to elements-1
                product = FPMul(Elem[D[n+r],e,esize], Elem[D[m+r],e,esize], StandardFPSCRValue());
                addend = if add then product else FPNeg(product);
                Elem[D[d+r],e,esize] = FPAAdd(Elem[D[d+r],e,esize], addend, StandardFPSCRValue());
    else // VFP instruction
        case esize of
            when 16
                addend16 = if add then FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]) else FPNeg(FPMul(S[n]<15:0>, S[m]<15:0>));
                S[d] = Zeros(16) : FPAAdd(S[d]<15:0>, addend16, FPSCR[]);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else FPNeg(FPMul(S[n], S[m], FPSCR[]));
                S[d] = FPAAdd(S[d], addend32, FPSCR[]);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else FPNeg(FPMul(D[n], D[m], FPSCR[]));
                D[d] = FPAAdd(D[d], addend64, FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VMLS{<c>}{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMLS{<c>}{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
add = (op == '0'); long_destination = FALSE;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMLSL (by scalar)

Vector Multiply Subtract Long multiplies elements of a vector by a scalar, and subtracts the products from corresponding elements of the destination vector. The destination vector elements are twice as long as the elements that are multiplied.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	!= 11	Vn	Vd	0	1	1	0	N	1	M	0	Vm										
size											op																				

A1

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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																				

T1

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); add = (op == '0'); floating_point = FALSE; long_destination = TRUE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                fp_addend = if add then FPMul(op1,op2,StandardFPSCRValue()) else FPNeg(FPMul(op1,op2,StandardFPSCRValue()));
                Elem[D[d+r],e,esize] = FPAdd(Elem[Din[d+r],e,esize], fp_addend, StandardFPSCRValue());
            else
                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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

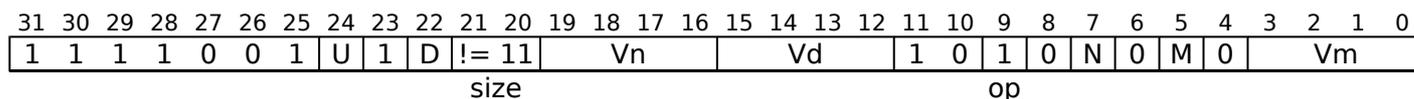
VMLSL (integer)

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



A1

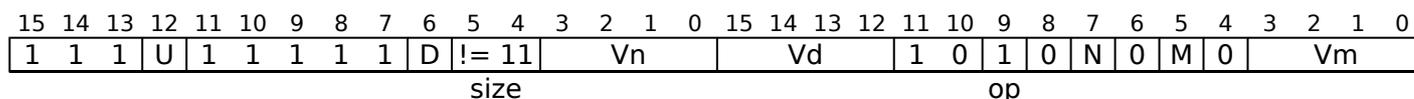
VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = 1;

```

T1



T1

VMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
add = (op == '0'); long_destination = TRUE; unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            product = Int(Elem[Din[n+r],e,esize],unsigned) * Int(Elem[Din[m+r],e,esize],unsigned);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

A1

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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			

T1

VMMLA{<q>}.BF16 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32BF16Ext() then UNDEFINED;
if Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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();  
bits(128) op1 = Q[n>>1];  
bits(128) op2 = Q[m>>1];  
bits(128) acc = Q[d>>1];  
Q[d>>1] = BFMatMulAdd(acc, op1, op2);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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																															

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

```
if !HaveFP16Ext() then UNDEFINED;
if cond != '1110' then UNPREDICTABLE;
to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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)

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>}.F16 <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>}.F16 <Rt>, <Sn>
```

```
if !HaveFP16Ext() then UNDEFINED;
if InITBlock() then UNPREDICTABLE;
to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

```
to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);  
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	0	0	op	Vn				Rt				1	0	1	0	N	(0)	(0)	1	(0)	(0)	(0)	(0)

From general-purpose register (op == 0)

```
VMOV{<c>}{<q>} <Sn>, <Rt>
```

To general-purpose register (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Sn>
```

```
to_arm_register = (op == '1'); t = UInt(Rt); n = UInt(Vn:N);  
if t == 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

- <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 [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

```
to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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			

From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Dm>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Dm>
```

```
to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(M:Vm);
if t == 15 || t2 == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (between two general-purpose registers and two single-precision registers)

Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers transfers the contents of two consecutively numbered single-precision Floating-point registers to two general-purpose registers, or the contents of two general-purpose registers to a pair of single-precision Floating-point registers. The general-purpose registers do not have to be contiguous.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	0	0	0	1	0	op	Rt2				Rt				1	0	1	0	0	0	M	1	Vm			
cond																															

From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

```
to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(Vm:M);
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			

From general-purpose registers (op == 0)

```
VMOV{<c>}{<q>} <Sm>, <Sm1>, <Rt>, <Rt2>
```

To general-purpose registers (op == 1)

```
VMOV{<c>}{<q>} <Rt>, <Rt2>, <Sm>, <Sm1>
```

```
to_arm_registers = (op == '1'); t = UInt(Rt); t2 = UInt(Rt2); m = UInt(Vm:M);  
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 Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .

Operation

```
if ConditionPassed() then  
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);  
    if to_arm_registers then  
        R[t] = S[m];  
        R[t2] = S[m+1];  
    else  
        S[m] = R[t];  
        S[m+1] = 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 (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																															

A1

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

```
boolean advsimd;
integer esize;
integer index;
case opc1:opc2 of
    when 'lxxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
    when '0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
    when '0x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
    when '0x10' UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt);
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	opc1	0	Vd				Rt				1	0	1	1	D	opc2	1	(0)	(0)	(0)	(0)		

T1

VMOV{<c>}{<q>}{.<size>} <Dd[x]>, <Rt>

```
boolean advsimd;
integer esize;
integer index;
case opc1:opc2 of
    when 'lxxx' advsimd = TRUE; esize = 8; index = UInt(opc1<0>:opc2);
    when '0xx1' advsimd = TRUE; esize = 16; index = UInt(opc1<0>:opc2<1>);
    when '0x00' advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
    when '0x10' UNDEFINED;
d = UInt(D:Vd); t = UInt(Rt);
if t == 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](#).

<size>	The data size. It must be one of:
8	Encoded as $\text{opc1}\langle 1 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle$, opc2 .
16	Encoded as $\text{opc1}\langle 1 \rangle = 0$, $\text{opc2}\langle 0 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle$, $\text{opc2}\langle 1 \rangle$.
32	Encoded as $\text{opc1}\langle 1 \rangle = 0$, $\text{opc2} = 0b00$. [x] is encoded in $\text{opc1}\langle 0 \rangle$.
omitted	Equivalent to 32.
<Dd[x]>	The scalar. The register <Dd> is encoded in D:Vd. For details of how [x] is encoded, see the description of <size>.
<Rt>	The source general-purpose register.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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											

64-bit SIMD vector (Q == 0)

```
VMOV{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMOV{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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																															

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VMOV{<c>}{<q>}.F16 <Sd>, #<imm>

Single-precision scalar (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

Double-precision scalar (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
integer d;
integer regs;
case size of
  when '01' d = UInt(Vd:D); imm16 = VFPEExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
  when '10' d = UInt(Vd:D); imm32 = VFPEExpandImm(imm4H:imm4L);
  when '11' d = UInt(D:Vd); imm64 = VFPEExpandImm(imm4H:imm4L); regs = 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.

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													

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

```

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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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													

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>

```

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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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			cmode				1	1	1	0	0	Q	1	1	imm4									
																							cmode							op										

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I64 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

```

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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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			cmode				0	x	x	0	0	Q	0	1	imm4									
																							cmode							op										

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I32 <Qd>, #<imm>

```

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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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			cmode				1	0	size	(0)	0	(0)	0	imm4L											
																							cmode							op										

**Half-precision scalar (size == 01)
(FEAT_FP16)**

VMOV{<c>}{<q>}.F16 <Sd>, #<imm>

Single-precision scalar (size == 10)

VMOV{<c>}{<q>}.F32 <Sd>, #<imm>

Double-precision scalar (size == 11)

VMOV{<c>}{<q>}.F64 <Dd>, #<imm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
single_register = (size != '11'); advsimd = FALSE;
bits(16) imm16;
bits(32) imm32;
bits(64) imm64;
integer d;
integer regs;
case size of
  when '01' d = UInt(Vd:D); imm16 = VFPEExpandImm(imm4H:imm4L); imm32 = Zeros(16) : imm16;
  when '10' d = UInt(Vd:D); imm32 = VFPEExpandImm(imm4H:imm4L);
  when '11' d = UInt(D:Vd); imm64 = VFPEExpandImm(imm4H:imm4L); regs = 1;

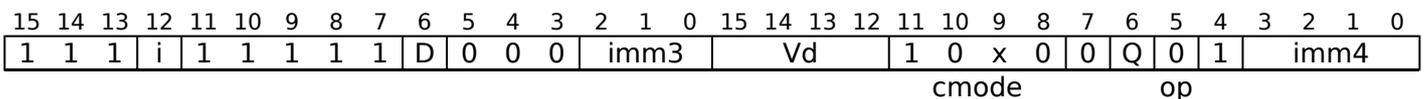
```

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.

T3



64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

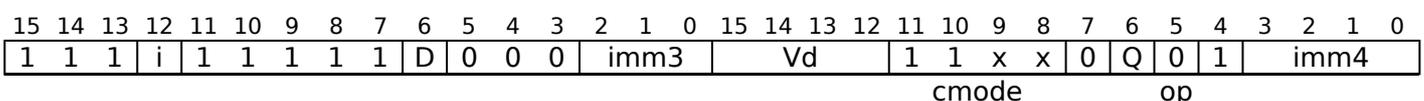
VMOV{<c>}{<q>}.I16 <Qd>, #<imm>

```

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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;

```

T4



64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.<dt> <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.<dt> <Qd>, #<imm>

```
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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;
```

T5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3			Vd			1 1 1 0 0			Q	1	1	imm4						
															cmode						op										

64-bit SIMD vector (Q == 0)

VMOV{<c>}{<q>}.I64 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMOV{<c>}{<q>}.I64 <Qd>, #<imm>

```
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;
single_register = FALSE; advsimd = TRUE; imm64 = AdvSIMDEExpandImm(op, cmode, i:imm3:imm4);
d = UInt(D:Vd); 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 single_register then
    S\[d\] = imm32;
  else
    for r = 0 to regs-1
      D\[d+r\] = 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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															

Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10'); advsimd = FALSE;
integer d;
integer m;
integer regs;
if single_register then
    d = UInt(Vd:D); m = UInt(Vm:M);
else
    d = UInt(D:Vd); m = UInt(M:Vm); regs = 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	1	1	0	1	D	1	1	0	0	0	0	Vd				1	0	1	x	0	1	M	0	Vm			
cond																size															

Single-precision scalar (size == 10)

```
VMOV{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VMOV{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
single_register = (size == '10'); advsimd = FALSE;
integer d;
integer m;
integer regs;
if single_register then
    d = UInt(Vd:D); m = UInt(Vm:M);
else
    d = UInt(D:Vd); m = UInt(M:Vm); regs = 1;
```

Assembler Symbols

<c> See [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```

if ConditionPassed\(\) then
    EncodingSpecificOperations(); CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if single_register then
        S[d] = S[m];
    else
        for r = 0 to regs-1
            D[d+r] = 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (register, SIMD)

Copy between SIMD registers copies the contents of one SIMD register to another

This is an alias of [VORR \(register\)](#). This means:

- The encodings in this description are named to match the encodings of [VORR \(register\)](#).
- The description of [VORR \(register\)](#) gives the operational pseudocode 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					

64-bit SIMD vector (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.

128-bit SIMD vector (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					

64-bit SIMD vector (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.

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOV (scalar to general-purpose register)

Copy a vector element to a general-purpose register with sign or zero extension copies a byte, halfword, or word from an Advanced SIMD scalar to a general-purpose register. Bytes and halfwords can be either zero-extended or sign-extended.

On a Floating-point-only system, this instruction transfers one word from the upper or lower half of a double-precision floating-point register to a general-purpose register. This is an identical operation to the Advanced SIMD single word transfer.

For more information about scalars see [Advanced SIMD scalars](#).

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
!= 1111				1	1	1	0	U	opc1	1	Vn				Rt				1	0	1	1	N	opc2				1	(0)	(0)	(0)	(0)
cond																																

A1

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

```
boolean advsimd;
integer esize;
integer index;
case U:opc1:opc2 of
  when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when 'x0xx1'  advsimd = TRUE;  esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '00x00'  advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '10x00'  UNDEFINED;
  when 'x0x10'  UNDEFINED;
t = UInt(Rt); n = UInt(N:Vn); unsigned = (U == '1');
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	U	opc1	1	Vn				Rt				1	0	1	1	N	opc2				1	(0)	(0)	(0)	(0)

T1

VMOV{<c>}{<q>}{.<dt>} <Rt>, <Dn[x]>

```
boolean advsimd;
integer esize;
integer index;
case U:opc1:opc2 of
  when 'x1xxx'  advsimd = TRUE;  esize = 8;  index = UInt(opc1<0>:opc2);
  when 'x0xx1'  advsimd = TRUE;  esize = 16; index = UInt(opc1<0>:opc2<1>);
  when '00x00'  advsimd = FALSE; esize = 32; index = UInt(opc1<0>);
  when '10x00'  UNDEFINED;
  when 'x0x10'  UNDEFINED;
t = UInt(Rt); n = UInt(N:Vn); unsigned = (U == '1');
if t == 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 .
<dt>	The data type. It must be one of: S8 Encoded as $U = 0, \text{opc1}\langle 1 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle, \text{opc2}$. S16 Encoded as $U = 0, \text{opc1}\langle 1 \rangle = 0, \text{opc2}\langle 0 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle, \text{opc2}\langle 1 \rangle$. U8 Encoded as $U = 1, \text{opc1}\langle 1 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle, \text{opc2}$. U16 Encoded as $U = 1, \text{opc1}\langle 1 \rangle = 0, \text{opc2}\langle 0 \rangle = 1$. [x] is encoded in $\text{opc1}\langle 0 \rangle, \text{opc2}\langle 1 \rangle$. 32 Encoded as $U = 0, \text{opc1}\langle 1 \rangle = 0, \text{opc2} = 0b00$. [x] is encoded in $\text{opc1}\langle 0 \rangle$. omitted Equivalent to 32.
<Rt>	The destination general-purpose register.
<Dn[x]>	The scalar. For details of how [x] is encoded see the description of <dt>.

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations(); CheckAdvSIMDorVFPEEnabled(TRUE, advsimd);
    if unsigned then
        R[t] = ZeroExtend(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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	0	M	1			Vm			

imm3H

A1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

```
if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1'); elements = 64 DIV esize;
d = UInt(D:Vd); 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	!= 000	0	0	0		Vd		1	0	1	0	0	0	0	M	1			Vm			

imm3H

T1

VMOVL{<c>}{<q>}.<dt> <Qd>, <Dm>

```
if imm3H == '000' then SEE "Related encodings";
if imm3H != '001' && imm3H != '010' && imm3H != '100' then SEE "VSHLL";
if Vd<0> == '1' then UNDEFINED;
esize = 8 * UInt(imm3H);
unsigned = (U == '1'); elements = 64 DIV esize;
d = UInt(D:Vd); 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](#).
- <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
        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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOVN

Vector Move and Narrow copies the least significant half of each element of a quadword vector into the corresponding elements of a doubleword vector.

The operand vector elements can be any one of 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the *CPACR*, *NSACR*, and *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*.

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	M	0	Vm							

A1

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); 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	M	0	Vm							

T1

VMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>

```
if size == '11' then UNDEFINED;
if Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMOVX

Vector Move extraction. This instruction copies the upper 16 bits of the 32-bit source SIMD&FP register into the lower 16 bits of the 32-bit destination SIMD&FP register, while clearing the remaining bits to zero.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_FP16)

31	30	29	28	27	26	25	24	23	22	21	20	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			

A1

VMOVX{<q>}.F16 <Sd>, <Sm>

```
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); 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			

T1

VMOVX{<q>}.F16 <Sd>, <Sm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveFP16Ext() then UNDEFINED;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
d = UInt(Vd:D); 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(); CheckVFPEnabled(TRUE);
    S[d] = Zeros(16) : S[m]<31:16>;
```


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 *ID group 0, Primary device identification registers*.

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																															

A1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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)

T1

```
VMRS{<c>}{<q>} <Rt>, <spec_reg>
```

```
t = UInt(Rt);
if !(reg IN {'000x', '0101', '011x', '1000'}) then UNPREDICTABLE;
if t == 15 && reg != '0001' then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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
        CheckVFPEnabled(TRUE);
        if t == 15 then
            PSTATE.<N,Z,C,V> = FPSR.<N,Z,C,V>;
        else
            R[t] = FPSCR;
    elsif PSTATE.EL == EL0 then
        UNDEFINED; // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(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

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMSR

Move general-purpose register to SIMD&FP Special register moves the value of a general-purpose register to a floating-point System register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

When these settings permit the execution of Advanced SIMD and floating-point instructions:

- If the specified floating-point System register is [FPSID](#) or [FPEXC](#), the instruction is UNDEFINED if executed in User mode.
- If the specified floating-point System register is the [FPSID](#) and the instruction is executed in a mode other than User mode, the instruction is ignored.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	1	1	0	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
cond																															

A1

VMSR{<c>}{<q>} <spec_reg>, <Rt>

```
t = UInt(Rt);
if !(reg IN {'000x'}) && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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)

T1

VMSR{<c>}{<q>} <spec_reg>, <Rt>

```
t = UInt(Rt);
if !(reg IN {'000x'}) && reg != '1000' then
    Constraint c = ConstrainUnpredictable(Unpredictable_VMSR);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
        when Constraint_UNDEF
            UNDEFINED;
        when Constraint_NOP
            EndOfInstruction();
if t == 15 then UNPREDICTABLE; // Armv8-A removes UNPREDICTABLE for R13
```

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
        CheckVFPEnabled(TRUE);
        FPSCR = R[t];
    elsif PSTATE.EL == EL0 then
        UNDEFINED; // Non-FPSCR registers accessible only at PL1 or above
    else
        CheckVFPEnabled(FALSE); // Non-FPSCR registers are not affected by FPEXC.EN
        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
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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

64-bit SIMD vector (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>[<index>]

```

if size == '11' then SEE "Related encodings";
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if size == '00' || (F == '1' && size == '01' && !HaveFP16Ext()) then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
floating_point = (F == '1'); long_destination = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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\(\);
  op2 = Elem\[Din\[m\],index,esize\]; op2val = Int(op2, unsigned);
  for r = 0 to regs-1
    for e = 0 to elements-1
      op1 = Elem\[Din\[n+r\],e,esize\]; op1val = Int(op1, unsigned);
      if floating_point then
        Elem\[D\[d+r\],e,esize\] = FPMul(op1, op2, StandardFPSCRValue\(\));
      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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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																															

Half-precision scalar (size == 01)
(FEAT_FP16)

VMUL{<c>}{<q>}.F16 {<Sd>, } <Sn>, <Sm>

Single-precision scalar (size == 10)

VMUL{<c>}{<q>}.F32 {<Sd>, } <Sn>, <Sm>

Double-precision scalar (size == 11)

VMUL{<c>}{<q>}.F64 {<Dd>, } <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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	0	D	0	sz	Vn				Vd				1	1	0	1	N	Q	M	1	Vm				

64-bit SIMD vector (Q == 0)

VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if sz == '1' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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						

Half-precision scalar (size == 01) (FEAT_FP16)

VMUL{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VMUL{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VMUL{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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
        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], StandardFPSCRVal);
    else // VFP instruction
        case esize of
            when 16
                S[d] = Zeros(16) : FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FPMul(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FPMul(D[n], D[m], FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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				

64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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				

64-bit SIMD vector (Q == 0)

```
VMUL{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMUL{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if size == '11' || (op == '1' && size != '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = FALSE; // "Don't care" value: TRUE produces same functionality
polynomial = (op == '1'); long_destination = FALSE;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize];    op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize];    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE; floating_point = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer index;
integer m;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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

T1

```
VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]
```

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
unsigned = (U == '1'); long_destination = TRUE; floating_point = FALSE;
d = UInt(D:Vd); n = UInt(N:Vn); regs = 1;
integer esize;
integer elements;
integer index;
integer m;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
    op2 = Elem[Din[m],index,esize]; op2val = Int(op2, unsigned);
    for r = 0 to regs-1
        for e = 0 to elements-1
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            if floating_point then
                Elem[D[d+r],e,esize] = FPMul(op1, op2, StandardFPSCRValue());
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VMULL (integer and polynomial)

Vector Multiply Long multiplies corresponding elements in two vectors. The destination vector elements are twice as long as the elements that are multiplied.

For information about multiplying polynomials see [Polynomial arithmetic over {0, 1}](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [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

A1

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then // .p64
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

T1

VMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
unsigned = (U == '1'); polynomial = (op == '1'); long_destination = TRUE;
esize = 8 << UInt(size); elements = 64 DIV esize;
if polynomial then
    if U == '1' || size == '01' then UNDEFINED;
    if size == '10' then // .p64
        if InITBlock() then UNPREDICTABLE;
        if !HaveBit128PMULLExt() then UNDEFINED;
        esize = 64; elements = 1;
if Vd<0> == '1' then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Elem[Din[n+r],e,esize]; op1val = Int(op1, unsigned);
            op2 = Elem[Din[m+r],e,esize]; 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I32 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I32 <Qd>, #<imm>

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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

64-bit SIMD vector (Q == 0)

VMVN{<c>}{<q>}.I16 <Dd>, #<imm>

128-bit SIMD vector (Q == 1)

VMVN{<c>}{<q>}.I16 <Qd>, #<imm>

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDEExpandImm('1', cmode, i:imm3:imm4);
d = UInt(D:Vd); regs = if Q == '0' then 1 else 2;
```

T3

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		1	1	0	x	0	Q	1	1		imm4					

cmode

64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}.I32 <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}.I32 <Qd>, #<imm>
```

```
if (cmode<0> == '1' && cmode<3:2> != '11') || cmode<3:1> == '111' then SEE "Related encodings";  
if Q == '1' && Vd<0> == '1' then UNDEFINED;  
imm64 = AdvSIMDExpandImm('1', cmode, i:imm3:imm4);  
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *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	0	Vd				0	1	0	1	1	Q	M	0	Vm				

64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (Q == 0)

```
VMVN{<c>}{<q>}{.<dt>} <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VMVN{<c>}{<q>}{.<dt>} <Qd>, <Qm>
```

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (Q == 0)

```
VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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

Half-precision scalar (size == 01) (FEAT_FP16)

```
VNEG{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VNEG{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VNEG{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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				

64-bit SIMD vector (Q == 0)

VNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

```

if size == '11' then UNDEFINED;
if F == '1' && ((size == '01' && !HaveFP16Ext()) || size == '00') then UNDEFINED;
if F == '1' && size == '01' && InITBlock() then UNPREDICTABLE;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
advsimd = TRUE; floating_point = (F == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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					

Half-precision scalar (size == 01) (FEAT_FP16)

VNEG{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VNEG{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VNEG{<c>}{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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
        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]);
                else
                    result = -SInt(Elem[D[m+r],e,esize]);
                    Elem[D[d+r],e,esize] = result<esize-1:0>;
    else
        // VFP instruction
        case esize of
            when 16 S[d] = Zeros(16) : FPNeg(S[m]<15:0>);
            when 32 S[d] = FPNeg(S[m]);
            when 64 D[d] = FPNeg(D[m]);
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VNMLA

Vector Negate Multiply Accumulate multiplies together two floating-point register values, adds the negation of the floating-point value in the destination register to the negation of the product, and writes the result back to the destination register.

Note

Arm recommends that software does not use the VNMLA instruction in the Round towards Plus Infinity and Round towards Minus Infinity rounding modes, because the rounding of the product and of the sum can change the result of the instruction in opposite directions, defeating the purpose of these rounding modes.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	1	Vn				Vd				1	0	size	N	1	M	0	Vm				
cond											op																				

Half-precision scalar (size == 01) (FEAT_FP16)

```
VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>
```

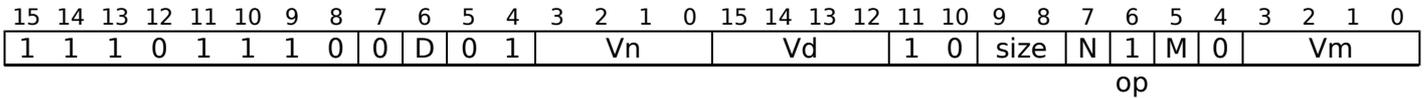
```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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



**Half-precision scalar (size == 01)
(FEAT_FP16)**

VNMLA{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMLA{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMLA{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

- <c> See [Standard assembler syntax fields](#).
- <q> See [Standard assembler syntax fields](#).
- <Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
- <Sn> Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
- <Sm> Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
- <Dd> Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
- <Dn> Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
- <Dm> Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    when VFPNegMul_VNMLS S[d] = FAdd(FPNeg(S[d]), product32, FPSCR[]);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    when VFPNegMul_VNMLS D[d] = FAdd(FPNeg(D[d]), product64, FPSCR[]);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	0	D	0	1	Vn				Vd				1	0	size	N	0	M	0	Vm				
cond												op																			

Half-precision scalar (size == 01) (FEAT_FP16)

VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && cond != '1110'`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	0	D	0	1	Vn				Vd				1	0	size	N	0	M	0	Vm				
																op															

Half-precision scalar (size == 01) (FEAT_FP16)

VNMLS{<c>}{<q>}.F16 <Sd>, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMLS{<c>}{<q>}.F32 <Sd>, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMLS{<c>}{<q>}.F64 <Dd>, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = if op == '1' then VFPNegMul_VNMLA else VFPNegMul_VNMLS;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    when VFPNegMul_VNMLS S[d] = FAdd(FPNeg(S[d]), product32, FPSCR[]);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    when VFPNegMul_VNMLS D[d] = FAdd(FPNeg(D[d]), product64, FPSCR[]);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Half-precision scalar (size == 01) (FEAT_FP16)

VNMUL{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VNMUL{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VNMUL{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```

if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
vtype = VFPNegMul_VNMUL;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);

```

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				

Half-precision scalar (size == 01) (FEAT_FP16)

```
VNMUL{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>
```

Single-precision scalar (size == 10)

```
VNMUL{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>
```

Double-precision scalar (size == 11)

```
VNMUL{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>
```

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '01' && !HaveFP16Ext() then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
vtype = VFPNegMul\_VNMUL;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    when VFPNegMul_VNMLS S[d] = FAdd(FPNeg(S[d]), product32, FPSCR[]);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    when VFPNegMul_VNMLS D[d] = FAdd(FPNeg(D[d]), product64, FPSCR[]);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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																															

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>}, <Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Dd>, #~<imm>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>}, <Dd>, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Dd>, #~<imm>

128-bit SIMD vector (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																															

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I16 {<Dd>}, {<Dd>}, #<imm>

is equivalent to

VORR{<c>}{<q>}.I16 <Dd>, #-<imm>

128-bit SIMD vector (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																			

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}.I32 {<Dd>}, {<Dd>}, #<imm>

is equivalent to

VORR{<c>}{<q>}.I32 <Dd>, #-<imm>

128-bit SIMD vector (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.

VORN (register)

Vector bitwise OR NOT (register) performs a bitwise OR NOT operation between two registers, and places the result in the destination register. The operand and result registers can be quadword or doubleword. They must all be the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	1	1	Vn				Vd				0	0	0	1	N	Q	M	1	Vm			

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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			

64-bit SIMD vector (Q == 0)

VORN{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VORN{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>}, {<Dd>}, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>}, {<Qd>}, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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							

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>}, {<Dd>}, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>}, {<Qd>}, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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							

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I32 {<Dd>}, <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I32 {<Qd>}, <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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															

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}.I16 {<Dd>}, <Dd>, #<imm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}.I16 {<Qd>}, <Qd>, #<imm>
```

```
if cmode<0> == '0' || cmode<3:2> == '11' then SEE "VMOV (immediate)";
if Q == '1' && Vd<0> == '1' then UNDEFINED;
imm64 = AdvSIMDExpandImm('0', cmode, i:imm3:imm4);
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;  
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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			

64-bit SIMD vector (Q == 0)

```
VORR{<c>}{<q>}{.<dt>} {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VORR{<c>}{<q>}{.<dt>} {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;  
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
VMOV (register, SIMD)	N:Vn == M:Vm
VRSHR (zero)	Never
VSHR (zero)	Never

Operation

```
if ConditionPassed\(\) then
    EncodingSpecificOperations(); CheckAdvSIMDEnabled\(\);
    for r = 0 to regs-1
        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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

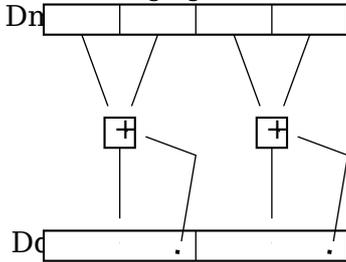
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADAL

Vector Pairwise Add and Accumulate Long adds adjacent pairs of elements of a vector, and accumulates the results into the elements of the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADAL doubleword operation for data type S16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0	Vd	0	1	1	0	op	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

VPADAL{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VPADAL{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize]; op2 = Elem[D[m+r],2*e+1,esize];
            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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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			

A1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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			

T1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if Q == '1' then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
    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], StandardFPSCRValue())
        Elem[dest,e+h,esize] = FPAdd(Elem[D[m],2*e,esize], Elem[D[m],2*e+1,esize], StandardFPSCRValue())

    D[d] = dest;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

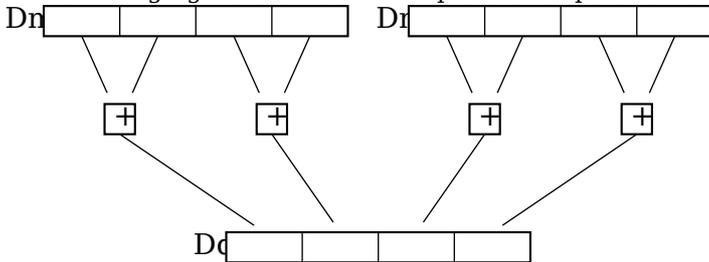
VPADD (integer)

Vector Pairwise Add (integer) adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The operands and result are doubleword vectors.

The operand and result elements must all be the same type, and can be 8-bit, 16-bit, or 32-bit integers. There is no distinction between signed and unsigned integers.

The following figure shows an example of the operation of VPADD doubleword operation for data type I16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size				Vn			Vd			1	0	1	1	N	Q	M	1			Vm	

A1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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	

T1

```
VPADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

```
if size == '11' || Q == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

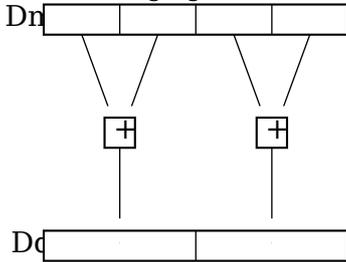
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPADDL

Vector Pairwise Add Long adds adjacent pairs of elements of two vectors, and places the results in the destination vector.

The vectors can be doubleword or quadword. The operand elements can be 8-bit, 16-bit, or 32-bit integers. The result elements are twice the length of the operand elements.

The following figure shows an example of the operation of VPADDL doubleword operation for data type S16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	0	0		Vd		0	0	1	0	op	Q	M	0		Vm				

64-bit SIMD vector (Q == 0)

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (Q == 0)

```
VPADDL{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VPADDL{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (op == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);
    h = elements DIV 2;

    for r = 0 to regs-1
        for e = 0 to h-1
            op1 = Elem[D[m+r],2*e,esize]; op2 = Elem[D[m+r],2*e+1,esize];
            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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

A1

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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

T1

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
  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 FPMax(op1,op2,StandardFPSCRValue\(\)) else FPMin(op1,op2,StandardFPSCRValue\(\));
    op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
    Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue\(\)) else FPMin(op1,op2,StandardFPSCRValue\(\));

  D[d] = dest;
```

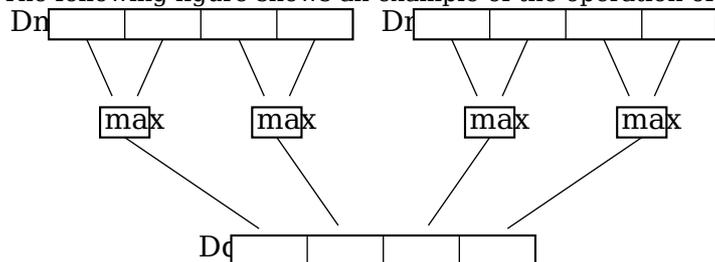
Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMAX (integer)

Vector Pairwise Maximum compares adjacent pairs of elements in two doubleword vectors, and copies the larger of each pair into the corresponding element in the destination doubleword vector.

The following figure shows an example of the operation of VPMAX doubleword operation for data type S16 or U16.



Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
1	1	1	1	0	0	1	U	0	D	size	Vn	Vd	1	0	1	0	N	0	M	0	Vm															
																					op															

A1

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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															

T1

VPMAX{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

A1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```

if sz == '1' && !HaveFP16Ext() then UNDEFINED;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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

T1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```

if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
maximum = (op == '0');
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
    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 FPMax(op1,op2,StandardFPSCRValue\(\)) else FPMin(op1,op2,StandardFPSCRValue\(\));
        op1 = Elem[D[m],2*e,esize]; op2 = Elem[D[m],2*e+1,esize];
        Elem[dest,e+h,esize] = if maximum then FPMax(op1,op2,StandardFPSCRValue\(\)) else FPMin(op1,op2,StandardFPSCRValue\(\));

    D[d] = dest;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPMIN (integer)

Vector Pairwise Minimum compares adjacent pairs of elements in two doubleword vectors, and copies the smaller of each pair into the corresponding element in the destination doubleword vector.

Depending on settings in the [CPACR](#), [NSACR](#), and [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	1		Vm	

op

A1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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	1		Vm	

op

T1

VPMIN{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

```
if size == '11' then UNDEFINED;
maximum = (op == '0'); unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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;
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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	1	0	0	1	D	1	1	1	1	0	1		Vd				1	0	1	1							0
cond				P		U	W		Rn				imm8<7:1>								imm8<0>										

Increment After

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
!=	1	1	1	1	1	0	0	1	D	1	1	1	1	0	1		Vd				1	0	1	0							imm8
cond				P		U	W		Rn				imm8																		

Increment After

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							0
				P		U	W		Rn				imm8<7:1>								imm8<0>										

Increment After

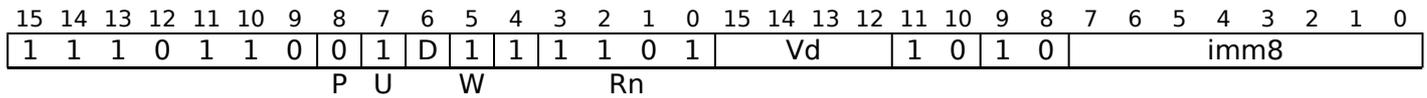
VPOP{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

[VLDM](#){<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2



Increment After

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VPUSH

Push SIMD&FP registers to stack stores multiple consecutive registers from the Advanced SIMD and floating-point register file to the stack.

This is an alias of [VSTM](#), [VSTMDB](#), [VSTMIA](#). This means:

- The encodings in this description are named to match the encodings of [VSTM](#), [VSTMDB](#), [VSTMIA](#).
- The description of [VSTM](#), [VSTMDB](#), [VSTMIA](#) gives the operational pseudocode 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>																		

Decrement Before

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				imm8																		

Decrement Before

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	W		Rn				imm8<0>																				

Decrement Before

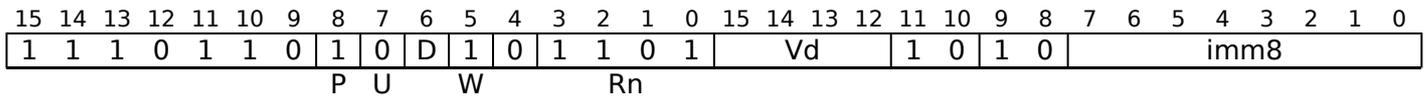
VPUSH{<c>}{<q>}{.<size>} <dreglist>

is equivalent to

[VSTMDB](#){<c>}{<q>}{.<size>} SP!, <dreglist>

and is always the preferred disassembly.

T2



Decrement Before

`VPU SH{<c>}{<q>}{.<size>} <sreglist>`

is equivalent to

`VSTM DB{<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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQABS

Vector Saturating Absolute takes the absolute value of each element in a vector, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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		

64-bit SIMD vector (Q == 0)

VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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		

64-bit SIMD vector (Q == 0)

VQABS{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VQABS{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
            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';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQADD

Vector Saturating Add adds the values of corresponding elements of two vectors, and places the results in the destination vector.

If any of the results overflow, they are saturated. The cumulative saturation bit, `FPSCR.QC`, is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the `CPACR`, `NSACR`, and `HCPTR` registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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										

64-bit SIMD vector (Q == 0)

```
VQADD{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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										

64-bit SIMD vector (Q == 0)

```
VQADD{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQADD{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            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';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 `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	0	1	D	!= 11		Vn		Vd		1	0	0	1	N	0	M	0					Vm			
size											op																				

A1

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;

```

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																				

A2

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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																				

T1

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;
```

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	1	D	!= 11			Vn			0	0	1	1	N	1	M	0			Vm					
										size							op														

T2

VQDMLAL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

A1

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;

```

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																				

A2

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

```

if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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																				

T1

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;
```

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	1	D	!= 11			Vn			0	1	1	1	N	1	M	0			Vm					
										size						op															

T2

VQDMLSL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>[<index>]

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
add = (op == '0');
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQDMULH

Vector Saturating Doubling Multiply Returning High Half multiplies corresponding elements in two vectors, doubles the results, and places the most significant half of the final results in the destination vector. The results are truncated, for rounded results see [VQRDMULH](#).

The second operand can be a scalar instead of a vector. For more information about scalars see [Advanced SIMD scalars](#).

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#) and [A2](#)) and T32 ([T1](#) and [T2](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	0	D	size	Vn	Vd	1	0	1	1	N	Q	M	0	Vm										

64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	Q	1	D	!= 11	Vn	Vd	1	1	0	0	N	1	M	0	Vm										

size

64-bit SIMD vector (Q == 0)

```
VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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				

64-bit SIMD vector (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	Q	1	1	1	1	1	D	!= 11	Vn					Vd					1	1	0	0	N	1	M	0	Vm				

size

64-bit SIMD vector (Q == 0)

VQDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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]);
      op1 = SInt(Elem[D[n+r],e,esize]);
      // The following only saturates if both op1 and op2 equal -(2^(esize-1))
      (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;
```

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

A2

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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										

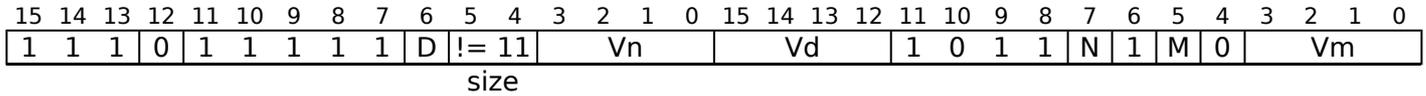
size

T1

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = FALSE; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
esize = 8 << UInt(size); elements = 64 DIV esize;
```

T2



T2

VQDMULL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' || Vd<0> == '1' then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn);
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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]);
        op1 = SInt(Elem[Din[n],e,esize]);
        // The following only saturates if both op1 and op2 equal -(2^(esize-1))
        (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

This instruction is used by the pseudo-instructions [VQRSHRN \(zero\)](#), [VQRSHRUN \(zero\)](#), [VQSHRN \(zero\)](#), and [VQSHRUN \(zero\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	1	0	op	M	0	Vm								

Signed result (op == 1x)

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11'); dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); 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								

Signed result (op == 1x)

```
VQMOVN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

Unsigned result (op == 01)

```
VQMOVUN{<c>}{<q>}.<dt> <Dd>, <Qm>
```

```
if op == '00' then SEE "VMOVN";
if size == '11' || Vm<0> == '1' then UNDEFINED;
src_unsigned = (op == '11'); dest_unsigned = (op<0> == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); 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
    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';
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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							

64-bit SIMD vector (Q == 0)

VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

VQNEG{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VQNEG{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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
      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';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VQRDMLAH

Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half. This instruction multiplies the vector elements of the first source SIMD&FP register with either the corresponding vector elements of the second source SIMD&FP register or the value of a vector element of the second source SIMD&FP register, without saturating the multiply results, doubles the results, and accumulates the most significant half of the final results with the vector elements of the destination SIMD&FP register. The results are rounded.

If any of the results overflow, they are saturated. The cumulative saturation bit, [FPSCR.QC](#), is set if saturation occurs. For details see [Pseudocode details of saturation](#).

Depending on settings in the [CPACR](#), [NSACR](#), and [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	

64-bit SIMD vector (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

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

64-bit SIMD vector (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```
if !HaveQRDMLAHExt() 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;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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						

64-bit SIMD vector (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if !HaveQRDMLAHExt() 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;
add = TRUE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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 (FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11		Vn		Vd		1	1	1	0	N	1	M	0		Vm						

size

64-bit SIMD vector (Q == 0)

VQRDMLAH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMLAH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```
if !HaveQRDMLAHExt() 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;
add = TRUE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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":

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\(\);
round_const = 1 << (esize-1);
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
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 + 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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										

64-bit SIMD vector (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if !HaveQRDMLAHExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); regs = if Q == '0' then 1 else 2;
```

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

64-bit SIMD vector (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```
if !HaveQRDMLAHExt() 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;
add = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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						

64-bit SIMD vector (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Qm>

```
if !HaveQRDMLAHExt() 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;
add = FALSE; scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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 (FEAT_RDM)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	Q	1	1	1	1	1	D	!= 11		Vn		Vd		1	1	1	1	N	1	M	0		Vm						

size

64-bit SIMD vector (Q == 0)

VQRDMLSH{<q>}.<dt> <Dd>, <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMLSH{<q>}.<dt> <Qd>, <Qn>, <Dm[x]>

```
if !HaveQRDMLAHExt() 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;
add = FALSE; scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer m;
integer index;
integer elements;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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":

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\(\);
round_const = 1 << (esize-1);
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
    op1 = SInt(Elem[D[n+r],e,esize]);
    op3 = SInt(Elem[D[d+r],e,esize]) << esize;
    if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
    (result, sat) = SignedSatQ((op3 - 2*(op1*op2) + round_const) >> esize, esize);
    Elem[D[d+r],e,esize] = result;
    if sat then FPSCR.QC = '1';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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										

64-bit SIMD vector (Q == 0)

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	Q	1	D	!= 11	Vn	Vd	1	1	0	1	N	1	M	0	Vm										

size

64-bit SIMD vector (Q == 0)

```
VQRDMULH{<c>}{<q>}.<dt> {<Dd>, } <Dn>, <Dm[x]>
```

128-bit SIMD vector (Q == 1)

```
VQRDMULH{<c>}{<q>}.<dt> {<Qd>, } <Qn>, <Dm[x]>
```

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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	0	D	size	Vn					Vd					1	0	1	1	N	Q	M	0	Vm				

64-bit SIMD vector (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '00' || size == '11' then UNDEFINED;
scalar_form = FALSE; esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	Q	1	1	1	1	1	D	!= 11	Vn					Vd					1	1	0	1	N	1	M	0	Vm				

size

64-bit SIMD vector (Q == 0)

VQRDMULH{<c>}{<q>}.<dt> {<Dd>, } <Dn>, <Dm[x]>

128-bit SIMD vector (Q == 1)

VQRDMULH{<c>}{<q>}.<dt> {<Qd>, } <Qn>, <Dm[x]>

```
if size == '11' then SEE "Related encodings";
if size == '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
scalar_form = TRUE; d = UInt(D:Vd); n = UInt(N:Vn); regs = if Q == '0' then 1 else 2;
integer esize;
integer elements;
integer m;
integer index;
if size == '01' then esize = 16; elements = 4; m = UInt(Vm<2:0>); index = UInt(M:Vm<3>);
if size == '10' then esize = 32; elements = 2; m = UInt(Vm); index = 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();
  round_const = 1 << (esize-1);
  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
      op1 = SInt(Elem[D[n+r],e,esize]);
      if !scalar_form then op2 = SInt(Elem[D[m+r],e,esize]);
      (result, sat) = SignedSatQ((2*op1*op2 + round_const) >> esize, esize);
      Elem[D[d+r],e,esize] = result;
      if sat then FPSCR.QC = '1';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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										

64-bit SIMD vector (Q == 0)

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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										

64-bit SIMD vector (Q == 0)

```
VQRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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

Signed result

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

Signed result

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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				

Signed result (!(imm6 == 000xxx) && op == 1)

```
VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

```
VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); 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				

Signed result (!(imm6 == 000xxx) && op == 1)

VQRSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

VQRSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VRSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); 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();
  round_const = 1 << (shift_amount - 1);
  for e = 0 to elements-1
    operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
    (result, sat) = SatQ((operand + round_const) >> shift_amount, 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 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	0	0	1	M	0	Vm				
op																															

Unsigned result

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	0	0	1	M	0	Vm				
op																															

Unsigned result

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.

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 [VQSRHL](#).

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										

64-bit SIMD vector (Q == 0)

```
VQSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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										

64-bit SIMD vector (Q == 0)

```
VQSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VQSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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
      shift = SInt(Elem[D[n+r],e,esize]<7:0>);
      operand = Int(Elem[D[m+r],e,esize], unsigned);
      (result,sat) = SatQ(operand << shift, esize, unsigned);
      Elem[D[d+r],e,esize] = result;
      if sat then FPSCR.QC = '1';
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	1	D	imm6						Vd			0	1	1	op	L	Q	M	1	Vm				

64-bit SIMD vector, signed result (!imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

```
VQSHL{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

64-bit SIMD vector, unsigned result (U == 1 && !imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

```
VQSHLU{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector, signed result (!imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

```
VQSHL{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

128-bit SIMD vector, unsigned result (U == 1 && !imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

```
VQSHLU{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
    when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
    when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
    when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
    when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector, signed result (!imm6 == 000xxx && L == 0) && op == 1 && Q == 0)

VQSHL{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

64-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 0)

VQSHLU{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>

128-bit SIMD vector, signed result (!imm6 == 000xxx && L == 0) && op == 1 && Q == 1)

VQSHL{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

128-bit SIMD vector, unsigned result (U == 1 && !(imm6 == 000xxx && L == 0) && op == 0 && Q == 1)

VQSHLU{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); m = UInt(M:Vm); 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
      operand = Int(Elem[D[m+r],e,esize], src_unsigned);
      (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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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

Signed result

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

Signed result

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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				

Signed result (!imm6 == 000xxx) && op == 1)

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result (U == 1 && !imm6 == 000xxx) && op == 0)

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
    when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
    when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
    when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); 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				

Signed result (!imm6 == 000xxx) && op == 1)

VQSHRN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

Unsigned result (U == 1 && !(imm6 == 000xxx) && op == 0)

VQSHRUN{<c>}{<q>}.<type><size> <Dd>, <Qm>, #<imm>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if U == '0' && op == '0' then SEE "VSHRN";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
src_unsigned = (U == '1' && op == '1'); dest_unsigned = (U == '1');
d = UInt(D:Vd); 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
    operand = Int(Elem[Qin[m>>1],e,2*esize], src_unsigned);
    (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 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																															

Unsigned result

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																															

Unsigned result

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.

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										

64-bit SIMD vector (Q == 0)

```
VQSUB{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQSUB{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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										

64-bit SIMD vector (Q == 0)

```
VQSUB{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VQSUB{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            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';

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

A1

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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

T1

VRADDHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = Elem[Qin[n>>1],e,2*esize] + Elem[Qin[m>>1],e,2*esize] + round_const;
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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							

64-bit SIMD vector (Q == 0)

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VRECPE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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();
  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], StandardFPSCRValue());
      else
        Elem[D[d+r],e,esize] = UnsignedRecipEstimate(Elem[D[m+r],e,esize]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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					

64-bit SIMD vector (Q == 0)

```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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					

64-bit SIMD vector (Q == 0)

```
VRECPS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRECPS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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]);
  
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

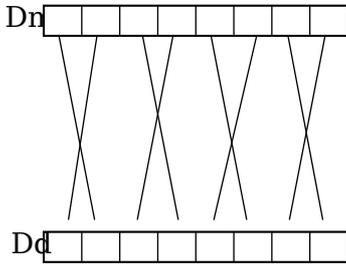
VREV16

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																															

64-bit SIMD vector (Q == 0)

VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>

```

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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																															

64-bit SIMD vector (Q == 0)

```
VREV16{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV16{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

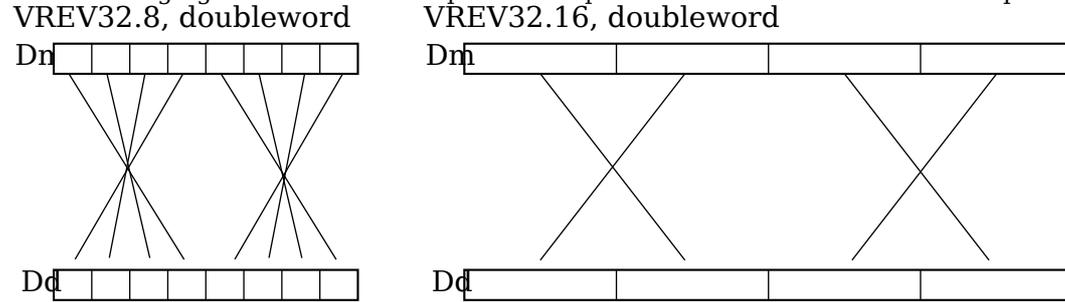
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	0	1	Q	M	0	Vm						

op

64-bit SIMD vector (Q == 0)

VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>

```

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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	1	Q	M	0	Vm						

op

64-bit SIMD vector (Q == 0)

```
VREV32{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV32{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

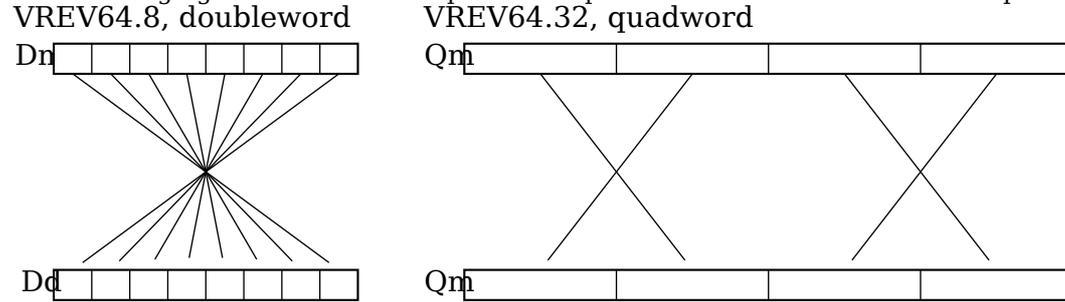
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	0	Q	M	0	Vm				
op																															

64-bit SIMD vector (Q == 0)

VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>

```

if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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																															

64-bit SIMD vector (Q == 0)

```
VREV64{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VREV64{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if UInt(op)+UInt(size) >= 3 then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;

esize = 8 << UInt(size);
integer container_size;
case op of
    when '10' container_size = 16;
    when '01' container_size = 32;
    when '00' container_size = 64;
integer containers = 64 DIV container_size;
integer elements_per_container = container_size DIV esize;

d = UInt(D:Vd); m = UInt(M:Vm); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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		

64-bit SIMD vector (Q == 0)

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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		

64-bit SIMD vector (Q == 0)

```
VRHADD{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRHADD{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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
            op1 = Int\(Elem\[D\[n+r\],e,esize\], unsigned\);
            op2 = Int\(Elem\[D\[m+r\],e,esize\], unsigned\);
            result = op1 + op2 + 1;
            Elem\[D\[d+r\],e,esize\] = result<esize:1>;

```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

64-bit SIMD vector (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

```

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VRINTA{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTA{<q>}.<dt> <Qd>, <Qm>

```

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' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
for r = 0 to regs-1  
  for e = 0 to elements-1  
    op1 = Elem[D[m+r],e,esize];  
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);  
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTA (floating-point)

Round floating-point to integer to Nearest with Ties to Away rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest with Ties to Away rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	0	0		Vd			1	0	!= 00	0	1	M	0			Vm		
														RM			size														

Half-precision scalar (size == 01)

(FEAT_FP16)

VRINTA{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTA{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTA{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); 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	0	1	D	1	1	1	0	0	0		Vd			1	0	!= 00	0	1	M	0			Vm		
														RM			size														

Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTA{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTA{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTA{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], 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

64-bit SIMD vector (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VRINTM{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTM{<q>}.<dt> <Qd>, <Qm>

```
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' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
for r = 0 to regs-1  
  for e = 0 to elements-1  
    op1 = Elem[D[m+r],e,esize];  
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);  
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

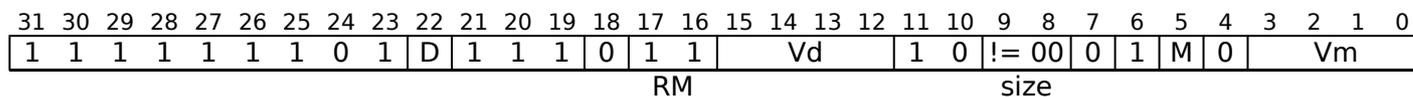
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Half-precision scalar (size == 01) (FEAT_FP16)

VRINTM{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTM{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

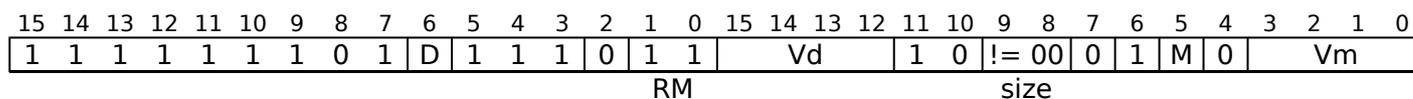
VRINTM{<q>}.F64 <Dd>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);

```

T1



Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTM{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTM{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTM{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], 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

64-bit SIMD vector (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

```

if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VRINTN{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTN{<q>}.<dt> <Qd>, <Qm>

```

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' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
for r = 0 to regs-1  
  for e = 0 to elements-1  
    op1 = Elem[D[m+r],e,esize];  
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);  
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

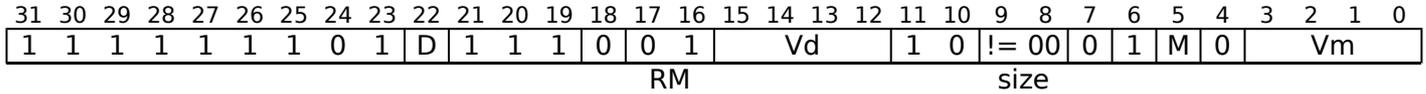
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

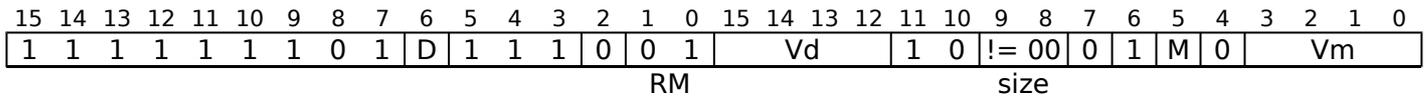
```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTN{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTN{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTN{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], 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

64-bit SIMD vector (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

```
if op<2> != op<0> then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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

64-bit SIMD vector (Q == 0)

VRINTP{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTP{<q>}.<dt> <Qd>, <Qm>

```
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' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
// Rounding encoded differently from other VCVT and VRINT instructions
rounding = FPDecodeRM(op<2>:NOT(op<1>)); exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
for r = 0 to regs-1  
  for e = 0 to elements-1  
    op1 = Elem[D[m+r],e,esize];  
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);  
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

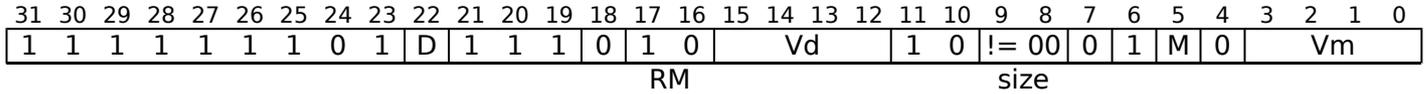
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1



Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

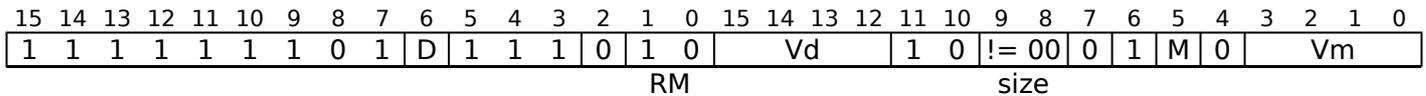
```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

T1



Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTP{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTP{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTP{<q>}.F64 <Dd>, <Dm>
```

```
if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
rounding = FPDecodeRM(RM); exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#).

Assembler Symbols

<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], 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																			

Half-precision scalar (size == 01) (FEAT_FP16)

VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); 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	0	1	D	1	1	0	1	1	0	Vd				1	0	size	0	1	M	0	Vm				
																op															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTR{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTR{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTR{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (Advanced SIMD)

Vector round floating-point to integer inexact rounds a vector of floating-point values to integral floating-point values of the same size, using the Round to Nearest rounding mode, and raises the Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	1	0	0	1	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

VRINTX{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTX{<q>}.<dt> <Qd>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN; exact = TRUE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

VRINTX{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTX{<q>}.<dt> <Qd>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_TIEEVEN; exact = TRUE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);
for r = 0 to regs-1
  for e = 0 to elements-1
    op1 = Elem[D[m+r],e,esize];
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);
    Elem[D[d+r],e,esize] = result;

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTX (floating-point)

Round floating-point to integer inexact rounds a floating-point value to an integral floating-point value of the same size, using the rounding mode specified in the FPSCR, and raises an Inexact exception when the result value is not numerically equal to the input value. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				1	1	1	0	1	D	1	1	0	1	1	1	Vd				1	0	size	0	1	M	0	Vm				

cond

Half-precision scalar (size == 01)

(FEAT_FP16)

VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
exact = TRUE;
integer esize;
integer d;
integer m;
case size of
    when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
    when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
    when '11' esize = 64; d = UInt(D:Vd); 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	0	1	D	1	1	0	1	1	1	Vd				1	0	size	0	1	M	0	Vm				

Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTX{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTX{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTX{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
exact = TRUE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEnabled(TRUE);
  rounding = FPRoundingMode(FPSCR[]);
  case esize of
    when 16
      S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTZ (Advanced SIMD)

Vector round floating-point to integer towards Zero rounds a vector of floating-point values to integral floating-point values of the same size, using the Round towards Zero rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic. It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	1	0	1	1	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTZ{<q>}.<dt> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO; exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

VRINTZ{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VRINTZ{<q>}.<dt> <Qd>, <Qm>

```

if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && !HaveFP16Ext()) || size IN {'00', '11'} then UNDEFINED;
rounding = FPRounding_ZERO; exact = FALSE;
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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\(\);  
for r = 0 to regs-1  
  for e = 0 to elements-1  
    op1 = Elem[D[m+r],e,esize];  
    result = FPRoundInt(op1, StandardFPSCRValue(), rounding, exact);  
    Elem[D[d+r],e,esize] = result;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																			

Half-precision scalar (size == 01) (FEAT_FP16)

VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>

Single-precision scalar (size == 10)

VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>

Double-precision scalar (size == 11)

VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); 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	0	1	D	1	1	0	1	1	0	Vd				1	0	size	1	1	M	0	Vm				
																op															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VRINTZ{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VRINTZ{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VRINTZ{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
exact = FALSE;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHL

Vector Rounding Shift Left takes each element in a vector, shifts them by a value from the least significant byte of the corresponding element of a second vector, and places the results in the destination vector. If the shift value is positive, the operation is a left shift. If the shift value is negative, it is a rounding right shift. For a truncating shift, see VSHL.

The first operand and result elements are the same data type, and can be any one of:

- 8-bit, 16-bit, 32-bit, or 64-bit signed integers.
- 8-bit, 16-bit, 32-bit, or 64-bit unsigned integers.

The second operand is always a signed integer of the same size.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	U	0	D	size	Vn	Vd	0	1	0	1	N	Q	M	0	Vm										

64-bit SIMD vector (Q == 0)

```
VRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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										

64-bit SIMD vector (Q == 0)

```
VRSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VRSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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			

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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			

64-bit SIMD vector (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

[VORR](#){<c>}{<q>}.<dt> <Dd>, <Dm>, <Dm>

128-bit SIMD vector (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			

64-bit SIMD vector (Q == 0)

VRSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

[VORR](#){<c>}{<q>}.<dt> <Dd>, <Dm>, <Dm>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector. For truncated results, see [VSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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				

A1 (imm6 != 000xxx)

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); 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				

T1 (imm6 != 000xxx)

```
VRSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); 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\(\);
    round_const = 1 << (shift_amount-1);
    for e = 0 to elements-1
        result = LSR(Elem[Qin[m>>1],e,2*esize] + round_const, shift_amount);
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRSHRN (zero)

Vector Rounding Shift Right and Narrow takes each element in a vector, right shifts them by an immediate value, and places the rounded results in the destination vector

This is a pseudo-instruction of [VMOVN](#). This means:

- The encodings in this description are named to match the encodings of [VMOVN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VMOVN](#) gives the operational pseudocode 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		

A1

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		

T1

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.

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							

64-bit SIMD vector (Q == 0)

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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							

64-bit SIMD vector (Q == 0)

```
VRSQRTE{<c>}{<q>}.<dt> <Dd>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTE{<c>}{<q>}.<dt> <Qd>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
if (size == '01' && (!HaveFP16Ext() || F == '0')) || size IN {'00', '11'} then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
integer esize;
integer elements;
case size of
    when '01' esize = 16; elements = 4;
    when '10' esize = 32; elements = 2;
d = UInt(D:Vd); m = UInt(M:Vm); 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();
  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], StandardFPSCRValue());
      else
        Elem[D[d+r],e,esize] = UnsignedRSqrtEstimate(Elem[D[m+r],e,esize]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	

64-bit SIMD vector (Q == 0)

```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

```
VRSQRTS{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VRSQRTS{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 [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	1	1	L	Q	M	1	Vm				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VRSRA{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VRSRA{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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();
    round_const = 1 << (shift_amount - 1);
    for r = 0 to regs-1
        for e = 0 to elements-1
            result = (Int(Elem[D[m+r],e,esize], unsigned) + round_const) >> 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

A1

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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

T1

VRSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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();
    round_const = 1 << (esize-1);
    for e = 0 to elements-1
        result = (Elem[Qin[n>>1],e,2*esize] - Elem[Qin[m>>1],e,2*esize]) + round_const;
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																													

64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]

128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]

```

if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
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																													

64-bit SIMD vector (Q == 0)

```
VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPEExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
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;
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;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSDOT (vector)

Dot Product vector form with signed integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

`ID_ISAR6`.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																								
1	1	1	1	1	1	0	0	0	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm																											
																												U																											

64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if !HaveD0TPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
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																											

64-bit SIMD vector (Q == 0)

VSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
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

```
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

Equal, half-precision scalar (cc == 00 && size == 01)
(FEAT_FP16)

VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Equal, single-precision scalar (cc == 00 && size == 10)

VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Equal, double-precision scalar (cc == 00 && size == 11)

VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Greater than or Equal, half-precision scalar (cc == 10 && size == 01)
(FEAT_FP16)

VSELGE.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Greater than or Equal, single-precision scalar (cc == 10 && size == 10)

VSELGE.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Greater than or Equal, double-precision scalar (cc == 10 && size == 11)

VSELGE.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Greater than, half-precision scalar (cc == 11 && size == 01)
(FEAT_FP16)

VSELGT.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Greater than, single-precision scalar (cc == 11 && size == 10)

VSELGT.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Greater than, double-precision scalar (cc == 11 && size == 11)

VSELGT.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

Unordered, half-precision scalar (cc == 01 && size == 01)
(FEAT_FP16)

VSELVS.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Unordered, single-precision scalar (cc == 01 && size == 10)

VSELVS.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)

Unordered, double-precision scalar (cc == 01 && size == 11)

VSELVS.F64 <Dd>, <Dn>, <Dm> // (Cannot be conditional)

```

if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = cc:(cc<1> EOR cc<0>):'0';

```

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	D	cc	Vn			Vd			1	0	!= 00	N	0	M	0	Vm							

size

Equal, half-precision scalar (cc == 00 && size == 01)
(FEAT_FP16)

VSELEQ.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Equal, single-precision scalar (cc == 00 && size == 10)

VSELEQ.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Equal, double-precision scalar (cc == 00 && size == 11)

VSELEQ.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Greater than or Equal, half-precision scalar (cc == 10 && size == 01)
(FEAT_FP16)

VSELGE.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Greater than or Equal, single-precision scalar (cc == 10 && size == 10)

VSELGE.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Greater than or Equal, double-precision scalar (cc == 10 && size == 11)

VSELGE.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Greater than, half-precision scalar (cc == 11 && size == 01)
(FEAT_FP16)

VSELGT.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Greater than, single-precision scalar (cc == 11 && size == 10)

VSELGT.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Greater than, double-precision scalar (cc == 11 && size == 11)

VSELGT.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

Unordered, half-precision scalar (cc == 01 && size == 01)
(FEAT_FP16)

VSELVS.F16 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Unordered, single-precision scalar (cc == 01 && size == 10)

VSELVS.F32 <Sd>, <Sn>, <Sm> // (Not permitted in IT block)

Unordered, double-precision scalar (cc == 01 && size == 11)

VSELVS.F64 <Dd>, <Dn>, <Dm> // (Not permitted in IT block)

```

if InITBlock() then UNPREDICTABLE;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
cond = 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
    S[d] = Zeros(16) : (if ConditionHolds(cond) then S[n] else S[m])<15:0>;
  when 32
    S[d] = if ConditionHolds(cond) then S[n] else S[m];
  when 64
    D[d] = if ConditionHolds(cond) then D[n] else D[m];

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHL (immediate)

Vector Shift Left (immediate) takes each element in a vector of integers, left shifts them by an immediate value, and places the results in the destination vector.

Bits shifted out of the left of each element are lost.

The elements must all be the same size, and can be 8-bit, 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	0	1	D	imm6						Vd			0	1	0	1	L	Q	M	1	Vm				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHL{<c>}{<q>}.I<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHL{<c>}{<q>}.I<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHL{<c>}{<q>}.I<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHL{<c>}{<q>}.I<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	

64-bit SIMD vector (Q == 0)

```
VSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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	

64-bit SIMD vector (Q == 0)

```
VSHL{<c>}{<q>}.<dt> {<Dd>}, <Dm>, <Dn>
```

128-bit SIMD vector (Q == 1)

```
VSHL{<c>}{<q>}.<dt> {<Qd>}, <Qm>, <Qn>
```

```
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1' || Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); n = UInt(N:Vn); 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
            shift = SInt(Elem[D[n+r],e,esize]<7:0>);
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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				

A1 (imm6 != 000xxx)

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '01xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '1xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1'); d = UInt(D:Vd); 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					

A2

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

```
if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = UInt(D:Vd); 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				

T1 (imm6 != 000xxx)

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

```

if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vd<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '01xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '1xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
if shift_amount == 0 then SEE "VMOVL";
unsigned = (U == '1'); d = UInt(D:Vd); 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				

T2

VSHLL{<c>}{<q>}.<type><size> <Qd>, <Dm>, #<imm>

```

if size == '11' || Vd<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize; shift_amount = esize;
unsigned = FALSE; // Or TRUE without change of functionality
d = UInt(D:Vd); 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
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSHR{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSHR{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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					

64-bit SIMD vector (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (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					

64-bit SIMD vector (Q == 0)

VSHR{<c>}{<q>}.<dt> <Dd>, <Dm>, #0

is equivalent to

VORR{<c>}{<q>}{.<dt>} <Dd>, <Dm>, <Dm>

128-bit SIMD vector (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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector. For rounded results, see [VRSHRN](#).

The operand elements can be 16-bit, 32-bit, or 64-bit integers. There is no distinction between signed and unsigned integers. The destination elements are half the size of the source elements.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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	M	1	Vm				

A1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); 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	M	1	Vm				

T1 (imm6 != 000xxx)

```
VSHRN{<c>}{<q>}.I<size> <Dd>, <Qm>, #<imm>
```

```
if imm6 IN {'000xxx'} then SEE "Related encodings";
if Vm<0> == '1' then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case imm6 of
  when '001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '01xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '1xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); 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
        result = LSR(Elem[Qin[m>>1],e,2*esize], shift_amount);
        Elem[D[d],e,esize] = result<esize-1:0>;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSHRN (zero)

Vector Shift Right Narrow takes each element in a vector, right shifts them by an immediate value, and places the truncated results in the destination vector

This is a pseudo-instruction of [VMOVN](#). This means:

- The encodings in this description are named to match the encodings of [VMOVN](#).
- The assembler syntax is used only for assembly, and is not used on disassembly.
- The description of [VMOVN](#) gives the operational pseudocode 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			

A1

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			

T1

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.

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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSLI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSLI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSLI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = UInt(imm6) - 8;
  when '001xxxx' esize = 16; elements = 4; shift_amount = UInt(imm6) - 16;
  when '01xxxxx' esize = 32; elements = 2; shift_amount = UInt(imm6) - 32;
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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();
    mask = LSL(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																					

A1

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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																					

T1

VSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSQRT

Square Root calculates the square root of the value in a floating-point register and writes the result to another floating-point register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [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																															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

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				

Half-precision scalar (size == 01) (FEAT_FP16)

```
VSQRT{<c>}{<q>}.F16 <Sd>, <Sm>
```

Single-precision scalar (size == 10)

```
VSQRT{<c>}{<q>}.F32 <Sd>, <Sm>
```

Double-precision scalar (size == 11)

```
VSQRT{<c>}{<q>}.F64 <Dd>, <Dm>
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
integer esize;
integer d;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); m = UInt(M:Vm);
```

CONSTRAINED UNPREDICTABLE behavior

If `size == '01' && InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16 S[d] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR[]);
    when 32 S[d] = FPSqrt(S[m], FPSCR[]);
    when 64 D[d] = FPSqrt(D[m], FPSCR[]);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRA{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRA{<c>}{<q>}.<type><size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRA{<c>}{<q>}.<type><size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
unsigned = (U == '1'); d = UInt(D:Vd); m = UInt(M:Vm); 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
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 [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	0	L	Q	M	1	Vm				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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	1	D	imm6						Vd			0	1	0	0	L	Q	M	1	Vm				

64-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 0)

```
VSRI{<c>}{<q>}.<size> {<Dd>}, <Dm>, #<imm>
```

128-bit SIMD vector (!(imm6 == 000xxx && L == 0) && Q == 1)

```
VSRI{<c>}{<q>}.<size> {<Qd>}, <Qm>, #<imm>
```

```
if (L:imm6) IN {'0000xxx'} then SEE "Related encodings";
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer esize;
integer elements;
integer shift_amount;
case L:imm6 of
  when '0001xxx' esize = 8; elements = 8; shift_amount = 16 - UInt(imm6);
  when '001xxxx' esize = 16; elements = 4; shift_amount = 32 - UInt(imm6);
  when '01xxxxx' esize = 32; elements = 2; shift_amount = 64 - UInt(imm6);
  when '1xxxxxx' esize = 64; elements = 1; shift_amount = 64 - UInt(imm6);
d = UInt(D:Vd); m = UInt(M:Vm); 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();
    mask = LSR(Ones(esize), shift_amount);
    for r = 0 to regs-1
        for e = 0 to elements-1
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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		

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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		

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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			

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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			

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 4;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 1; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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				

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 2; if align == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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											0	1	1	0	size	align		Rm

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
regs = 3; if align<1> == '1' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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.

{ <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]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to regs-1
        for e = 0 to elements-1
            if ebytes != 8 then
                MemU[address,ebytes] = Elem[D[d+r],e];
            else
                - = AArch32.CheckAlignment(address, ebytes, AccType_NORMAL, iswrite);
                bits(64) data = Elem[D[d+r],e];
                MemU[address,4] = if BigEndian(AccType_NORMAL) then data<63:32> else data<31:0>;
                MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																															

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
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;
ebytes = 4; index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
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	1	1	0	0	1	1	D	0	0		Rn		Vd					0	0	0	0	index_align					Rm		
size																															

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); alignment = 1;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	1	D	0	0		Rn		Vd					0	1	0	0	index_align					Rm		
size																															

Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

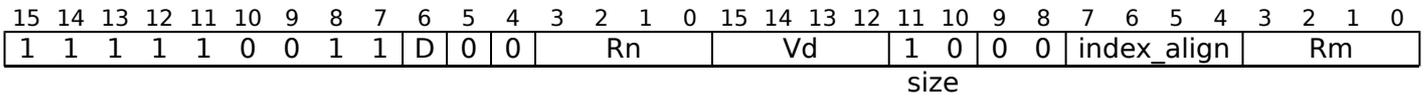
VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 then UNPREDICTABLE;

```

T3



Offset (Rm == 1111)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST1{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

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;
ebytes = 4; index = UInt(index_align<3>);
alignment = if index_align<1:0> == '00' then 1 else 4;
d = UInt(D:Vd); n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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();
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    MemU[address,ebytes] = Elem[D[d],index];
    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.

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

Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```

pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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	

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

pairs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																		

itype

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

pairs = 1; if align == '11' then UNDEFINED;
if size == '11' then UNDEFINED;
inc = if itype == '1001' then 2 else 1;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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							

Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
pairs = 2; inc = 2;
if size == '11' then UNDEFINED;
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for r = 0 to pairs-1
        for e = 0 to elements-1
            MemU[address, ebytes] = Elem[D[d+r], e];
            MemU[address+ebytes, ebytes] = Elem[D[d2+r], e];
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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 *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	0	0				Rn				Vd			0	0	0	1	index_align			Rm		

size

Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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																															

Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

```

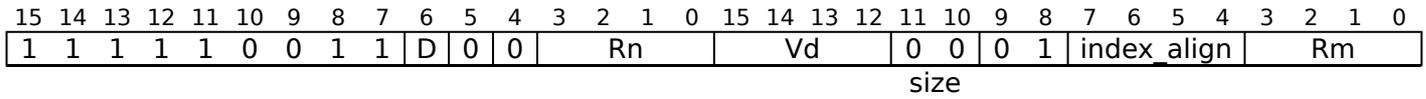
CONSTRAINED UNPREDICTABLE behavior

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



Offset (Rm == 1111)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

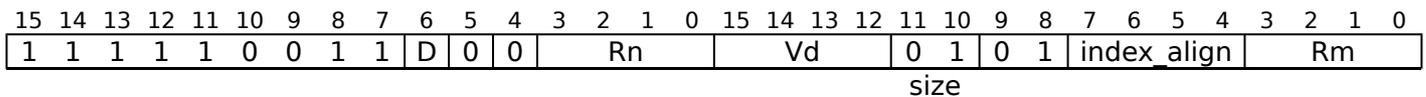
```
if size == '11' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;
```

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



Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d2 > 31 then UNPREDICTABLE;

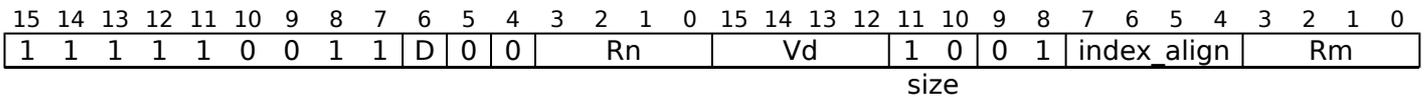
```

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



Offset (Rm == 1111)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST2{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<1> != '0' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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\(\);
  address = R\[n\]; iswrite = TRUE;
  - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
  MemU[address, ebytes] = Elem\[D\[d\], index\];
  MemU[address+ebytes, ebytes] = Elem\[D\[d2\], index\];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

Offset (Rm == 1111)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' || align<l> == '1' then UNDEFINED;
integer inc;
case itype of
    when '0100'
        inc = 1;
    when '0101'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0				Rn				Vd			0	1	0	x	size	align			Rm	

itype

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
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";
alignment = if align<0> == '0' then 1 else 8;
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = TRUE;
  - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
  for e = 0 to elements-1
    MemU[address, ebytes] = Elem[D[d], e];
    MemU[address+ebytes, ebytes] = Elem[D[d2], e];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], e];
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
 Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VST3 (single 3-element structure from one lane)

Store single 3-element structure from one lane of three registers stores one 3-element structure to memory from corresponding elements of three registers. For details of the addressing mode, see *Advanced SIMD addressing mode*. Depending on settings in the *CPACR*, *NSACR*, and *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	0	0				Rn				Vd			0	0	1	0	index_align				Rm	

size

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```
if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	0	1	D	0	0																				
												Rn		Vd		1 0		1 0		index_align		Rm									
																		size													

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

```

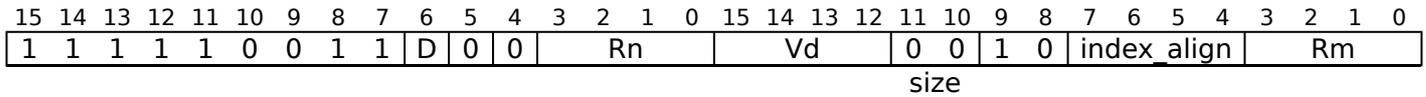
CONSTRAINED UNPREDICTABLE behavior

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



Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>!]

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); register_index = (m != 15 && m != 13);
if n == 15 || d3 > 31 then UNPREDICTABLE;

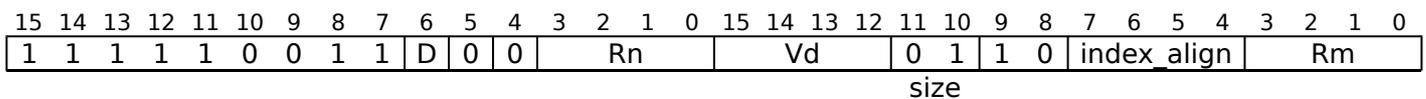
```

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



Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<0> != '0' then UNDEFINED;
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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	0	index_align		Rm					
																size															

Offset (Rm == 1111)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]

Post-indexed (Rm == 1101)

VST3{<c>}{<q>}.<size> <list>, [<Rn>]!

Post-indexed (Rm != 11x1)

VST3{<c>}{<q>}.<size> <list>, [<Rn>], <Rm>

```

if size == '11' then UNDEFINED;
if index_align<1:0> != '00' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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(); CheckAdvSIMDEnabled\(\);
  address = R[n];
  MemU[address, ebytes] = Elem[D[d], index];
  MemU[address+ebytes, ebytes] = Elem[D[d2],index];
  MemU[address+2*ebytes,ebytes] = Elem[D[d3],index];
  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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 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

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
integer inc;
case itype of
    when '0000'
        inc = 1;
    when '0001'
        inc = 2;
    otherwise
        SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
integer inc;
case itype of
  when '0000'
    inc = 1;
  when '0001'
    inc = 2;
  otherwise
    SEE "Related encodings";
alignment = if align == '00' then 1 else 4 << UInt(align);
ebytes = 1 << UInt(size); elements = 8 DIV ebytes;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType_VEC, iswrite);
    for e = 0 to elements-1
        MemU[address, ebytes] = Elem[D[d], e];
        MemU[address+ebytes, ebytes] = Elem[D[d2], e];
        MemU[address+2*ebytes, ebytes] = Elem[D[d3], e];
        MemU[address+3*ebytes, ebytes] = Elem[D[d4], e];
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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

Offset (Rm == 1111)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]
```

Post-indexed (Rm == 1101)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!
```

Post-indexed (Rm != 11x1)

```
VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>
```

```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```
if size == '11' then UNDEFINED;
if size != '00' then SEE "Related encodings";
ebytes = 1; index = UInt(index_align<3:1>); inc = 1;
alignment = if index_align<0> == '0' then 1 else 4;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

```

if size == '11' then UNDEFINED;
if size != '01' then SEE "Related encodings";
ebytes = 2; index = UInt(index_align<3:2>);
inc = if index_align<1> == '0' then 1 else 2;
alignment = if index_align<0> == '0' then 1 else 8;
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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															

Offset (Rm == 1111)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]

Post-indexed (Rm != 11x1)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}], <Rm>

Post-indexed (Rm == 1101)

VST4{<c>}{<q>}.<size> <list>, [<Rn>{:<align>}]!

```

if size == '11' then UNDEFINED;
if size != '10' then SEE "Related encodings";
if index_align<1:0> == '11' then UNDEFINED;
ebytes = 4; index = UInt(index_align<3>);
inc = if index_align<2> == '0' then 1 else 2;
alignment = if index_align<1:0> == '00' then 1 else 4 << UInt(index_align<1:0>);
d = UInt(D:Vd); d2 = d + inc; d3 = d2 + inc; d4 = d3 + inc; n = UInt(Rn); m = UInt(Rm);
wback = (m != 15); 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\(\);
    address = R[n]; iswrite = TRUE;
    - = AArch32.CheckAlignment(address, alignment, AccType\_VEC, iswrite);
    MemU[address,          ebytes] = Elem[D[d], index];
    MemU[address+ebytes,  ebytes] = Elem[D[d2], index];
    MemU[address+2*ebytes, ebytes] = Elem[D[d3], index];
    MemU[address+3*ebytes, ebytes] = Elem[D[d4], index];
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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>			

Decrement Before (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

```

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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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																															

Decrement Before (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>

Increment After (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>

```

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 !)
single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); 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>				

Decrement Before (P == 1 && U == 0 && W == 1)

VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <dreglist>

Increment After (P == 0 && U == 1)

VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <dreglist>

```

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 !)
single_regs = FALSE; add = (U == '1'); wback = (W == '1');
d = UInt(D:Vd); n = UInt(Rn); imm32 = ZeroExtend(imm8:'00', 32);
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							

Decrement Before (P == 1 && U == 0 && W == 1)

```
VSTMDB{<c>}{<q>}{.<size>} <Rn>!, <sreglist>
```

Increment After (P == 0 && U == 1)

```
VSTM{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
VSTMIA{<c>}{<q>}{.<size>} <Rn>{!}, <sreglist>
```

```
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 !)
single_regs = TRUE; add = (U == '1'); wback = (W == '1'); d = UInt(Vd:D); n = UInt(Rn);
imm32 = ZeroExtend(imm8:'00', 32); 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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

<size> An optional data size specifier. If present, it must be equal to the size in bits, 32 or 64, of the registers being transferred.

<Rn> Is the general-purpose base register, encoded in the "Rn" field. If writeback is not specified, the PC can be used. However, Arm deprecates use of the PC.

- ! 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
VPUSH	<code>P == '1' && U == '0' && W == '1' && 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
      MemA[address,4] = S[d+r]; address = address+4;
    else
      // Store as two word-aligned words in the correct order for current endianness.
      MemA[address,4] = if BigEndian(AccType\_ATOMIC) then D[d+r]<63:32> else D[d+r]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType\_ATOMIC) then D[d+r]<31:0> else 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																															

Half-precision scalar (size == 01) (FEAT_FP16)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #<+/-><imm>}]
```

Single-precision scalar (size == 10)

```
VSTR{<c>}{<q>}.32 <Sd>, [<Rn>{, #<+/-><imm>}]
```

Double-precision scalar (size == 11)

```
VSTR{<c>}{<q>}.64 <Dd>, [<Rn>{, #<+/-><imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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								

Half-precision scalar (size == 01) (FEAT_FP16)

```
VSTR{<c>}{<q>}.16 <Sd>, [<Rn>{, #<+/-><imm>}]
```

Single-precision scalar (size == 10)

```
VSTR{<c>}{<q>}.32 <Sd>, [<Rn>{, #<+/-><imm>}]
```

Double-precision scalar (size == 11)

```
VSTR{<c>}{<q>}.64 <Dd>, [<Rn>{, #<+/-><imm>}]
```

```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
esize = 8 << UInt(size); add = (U == '1');
imm32 = if esize == 16 then ZeroExtend(imm8:'0', 32) else ZeroExtend(imm8:'00', 32);
integer d;
case size of
  when '01' d = UInt(Vd:D);
  when '10' d = UInt(Vd:D);
  when '11' d = UInt(D:Vd);
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 [Standard assembler syntax fields](#).

<q> See [Standard assembler syntax fields](#).

.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":

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(); CheckVFPEnabled(TRUE);
  address = if add then (R[n] + imm32) else (R[n] - imm32);
  case esize of
    when 16
      MemA[address,2] = S[d]<15:0>;
    when 32
      MemA[address,4] = S[d];
    when 64
      // Store as two word-aligned words in the correct order for current endianness.
      MemA[address,4] = if BigEndian(AccType\_ATOMIC) then D[d]<63:32> else D[d]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType\_ATOMIC) then D[d]<31:0> else 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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	

64-bit SIMD vector (Q == 0)

```
VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>
```

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
    when '0' esize = 32; elements = 2;
    when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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

Half-precision scalar (size == 01)
(FEAT_FP16)

VSUB{<c>}{<q>}.F16 {<Sd>, } <Sn>, <Sm>

Single-precision scalar (size == 10)

VSUB{<c>}{<q>}.F32 {<Sd>, } <Sn>, <Sm>

Double-precision scalar (size == 11)

VSUB{<c>}{<q>}.F64 {<Dd>, } <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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					

64-bit SIMD vector (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE;
integer esize;
integer elements;
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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						

Half-precision scalar (size == 01) (FEAT_FP16)

VSUB{<c>}{<q>}.F16 {<Sd>}, <Sn>, <Sm>

Single-precision scalar (size == 10)

VSUB{<c>}{<q>}.F32 {<Sd>}, <Sn>, <Sm>

Double-precision scalar (size == 11)

VSUB{<c>}{<q>}.F64 {<Dd>}, <Dn>, <Dm>

```
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
advsimd = FALSE;
integer esize;
integer d;
integer n;
integer m;
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  when '11' esize = 64; d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm);
```

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
    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], StandardFPSCRVal)
  else // VFP instruction
    case esize of
      when 16
        S[d] = Zeros(16) : FPSub(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
      when 32
        S[d] = FPSub(S[n], S[m], FPSCR[]);
      when 64
        D[d] = FPSub(D[n], D[m], FPSCR[]);

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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						

64-bit SIMD vector (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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						

64-bit SIMD vector (Q == 0)

VSUB{<c>}{<q>}.<dt> {<Dd>, }<Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VSUB{<c>}{<q>}.<dt> {<Qd>, }<Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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

A1

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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

T1

VSUBHN{<c>}{<q>}.<dt> <Dd>, <Qn>, <Qm>

```
if size == '11' then SEE "Related encodings";
if Vn<0> == '1' || Vm<0> == '1' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); 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
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																				

A1

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  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	!= 11		Vn		Vd		0	0	1	0	N	0	M	0					Vm			
size											op																				

T1

VSUBL{<c>}{<q>}.<dt> <Qd>, <Dn>, <Dm>

```
if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  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 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);
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUBW

Vector Subtract Wide subtracts the elements of a doubleword vector from the corresponding elements of a quadword vector, and places the results in another quadword vector. Before subtracting, it sign-extends or zero-extends the elements of the doubleword operand.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	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																				

A1

VSUBW{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  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	!= 11	Vn	Vd	0	0	1	1	N	0	M	0	Vm										
size											op																				

T1

VSUBW{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Dm>

```

if size == '11' then SEE "Related encodings";
if Vd<0> == '1' || (op == '1' && Vn<0> == '1') then UNDEFINED;
unsigned = (U == '1');
esize = 8 << UInt(size);  elements = 64 DIV esize;  is_vsubw = (op == '1');
d = UInt(D:Vd);  n = UInt(N:Vn);  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 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);
        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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSUDOT (by element)

Dot Product index form with signed and unsigned integers. This instruction performs the dot product of the four signed 8-bit integer values in each 32-bit element of the first source register with the four unsigned 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. [ID_ISAR6.I8MM](#) indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	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	1	Vm																															
																												U																															

64-bit SIMD vector (Q == 0)

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
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	1	Vm																															
																												U																															

64-bit SIMD vector (Q == 0)

```
VSUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VSUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
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;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VSWP

Vector Swap exchanges the contents of two vectors. The vectors can be either doubleword or quadword. There is no distinction between data types.

Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	0	0	1	0	Vd				0	0	0	0	0	Q	M	0	Vm			
size																															

64-bit SIMD vector (Q == 0)

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); 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																															

64-bit SIMD vector (Q == 0)

VSWP{<c>}{<q>}{.<dt>} <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VSWP{<c>}{<q>}{.<dt>} <Qd>, <Qm>

```
if size != '00' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
d = UInt(D:Vd); m = UInt(M:Vm); 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
        if d == m then
             $D[d+r] = \text{bits}(64) \text{ UNKNOWN};$ 
        else
             $D[d+r] = D_{in}[m+r];$ 
             $D[m+r] = D_{in}[d+r];$ 
```

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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTBL, VTBX

Vector Table Lookup uses byte indexes in a control vector to look up byte values in a table and generate a new vector. Indexes out of range return 0.

Vector Table Extension works in the same way, except that indexes out of range leave the destination element unchanged.

Depending on settings in the *CPACR*, *NSACR*, and *HCPTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1			Vn			Vd			1	0	len	N	op	M	0			Vm		

VTBL (op == 0)

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

VTBX (op == 1)

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

```
is_vtbl = (op == '0'); length = UInt(len)+1;
d = UInt(D:Vd); n = UInt(N:Vn); 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.

T1

15	14	13	12	11	10	9	8	7	6	5	4	3	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		

VTBL (op == 0)

```
VTBL{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

VTBX (op == 1)

```
VTBX{<c>}{<q>}.8 <Dd>, <list>, <Dm>
```

```
is_vtbl = (op == '0'); length = UInt(len)+1;
d = UInt(D:Vd); n = UInt(N:Vn); 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 [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.
- <list> The vectors containing the table. It must be one of:
{<Dn>}
 Encoded as len = 0b00.
{<Dn>, <Dn+1>}
 Encoded as len = 0b01.
{<Dn>, <Dn+1>, <Dn+2>}
 Encoded as len = 0b10.
{<Dn>, <Dn+1>, <Dn+2>, <Dn+3>}
 Encoded as len = 0b11.
- <Dm> Is the 64-bit name of the SIMD&FP source register holding the indices, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations();  CheckAdvSIMDEnabled();

  // Create 256-bit = 32-byte table variable, with zeros in entries that will not be used.
  table3 = if length == 4 then D[n+3] else Zeros(64);
  table2 = if length >= 3 then D[n+2] else Zeros(64);
  table1 = if length >= 2 then D[n+1] else Zeros(64);
  table = table3 : table2 : table1 : D[n];

  for i = 0 to 7
    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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

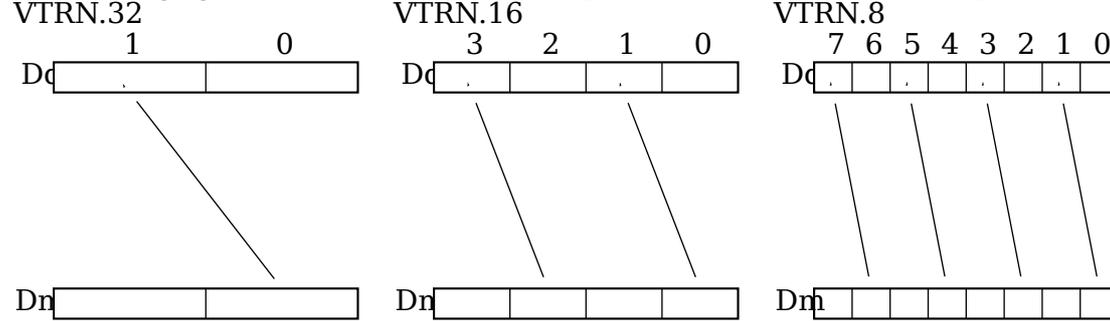
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VTRN

Vector Transpose treats the elements of its operand vectors as elements of 2 x 2 matrices, and transposes the matrices.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VTRN doubleword operations.



Depending on settings in the [CPACR](#), [NSACR](#), and [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

This instruction is used by the pseudo-instructions [VUZP \(alias\)](#), and [VZIP \(alias\)](#).

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0		Vd		0	0	0	0	1	Q	M	0		Vm				

64-bit SIMD vector (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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				

64-bit SIMD vector (Q == 0)

VTRN{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VTRN{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); m = UInt(M:Vm); 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();
    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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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	

64-bit SIMD vector (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VTST{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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	

64-bit SIMD vector (Q == 0)

VTST{<c>}{<q>}.<dt> {<Dd>}, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VTST{<c>}{<q>}.<dt> {<Qd>}, <Qn>, <Qm>

```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if size == '11' then UNDEFINED;
esize = 8 << UInt(size); elements = 64 DIV esize;
d = UInt(D:Vd); n = UInt(N:Vn); m = UInt(M:Vm); 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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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																													

64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
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																													

64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPEExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean signed = (U=='0');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm<3:0>);
integer index = UInt(M);
integer esize = 32;
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;
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;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUDOT (vector)

Dot Product vector form with unsigned integers. This instruction performs the dot product of the four 8-bit elements in each 32-bit element of the first source register with the four 8-bit elements of the corresponding 32-bit element in the second source register, accumulating the result into the corresponding 32-bit element of the destination register. In Armv8.2 and Armv8.3, this is an OPTIONAL instruction. From Armv8.4 it is mandatory for all implementations to support it.

Note

`ID_ISAR6`.DP indicates whether this instruction is supported.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_DotProd)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	0	D	1	0		Vn		Vd		1	1	0	1	N	Q	M	1						Vm	
																											U				

64-bit SIMD vector (Q == 0)

VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>

```
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
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

64-bit SIMD vector (Q == 0)

```
VUDOT{<q>}.U8 <Dd>, <Dn>, <Dm>
```

128-bit SIMD vector (Q == 1)

```
VUDOT{<q>}.U8 <Qd>, <Qn>, <Qm>
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveDOTPExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
boolean signed = U=='0';
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
integer esize = 32;
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

```
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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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																			

A1

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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																			

T1

VUMMLA{<q>}.U8 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (by element)

Dot Product index form with unsigned and signed integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in an indexed 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. [ID_ISAR6.I8MM](#) indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	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																															

64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
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																															

64-bit SIMD vector (Q == 0)

```
VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>[<index>]
```

128-bit SIMD vector (Q == 1)

```
VUSDOT{<q>}.S8 <Qd>, <Qn>, <Dm>[<index>]
```

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1') then UNDEFINED;
boolean op1_unsigned = (U == '0');
boolean op2_unsigned = (U == '1');
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(Vm);
integer i = UInt(M);
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;
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSDOT (vector)

Dot Product vector form with mixed-sign integers. This instruction performs the dot product of the four unsigned 8-bit integer values in each 32-bit element of the first source register with the four signed 8-bit integer values in the corresponding 32-bit element of the second source register, accumulating the result into the corresponding 32-bit element of the destination register.

From Armv8.2, this is an OPTIONAL instruction. [ID_ISAR6.I8MM](#) indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	0	0	1	D	1	0	Vn				Vd				1	1	0	1	N	Q	M	0	Vm			

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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			

64-bit SIMD vector (Q == 0)

VUSDOT{<q>}.S8 <Dd>, <Dn>, <Dm>

128-bit SIMD vector (Q == 1)

VUSDOT{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
integer m = UInt(M:Vm);
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
      element1 = UInt(Elem[operand1, 4 * e + b, 8]);
      element2 = SInt(Elem[operand2, 4 * e + b, 8]);
      res = res + element1 * element2;
    Elem[result, e, 32] = res;
  D[d+r] = result;

```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUSMMLA

The widening integer matrix multiply-accumulate instruction multiplies the 2x8 matrix of unsigned 8-bit integer values held in the first source vector by the 8x2 matrix of signed 8-bit integer values in the second source vector. The resulting 2x2 32-bit integer matrix product is destructively added to the 32-bit integer matrix accumulator held in the destination vector. This is equivalent to performing an 8-way dot product per destination element.

From Armv8.2, this is an OPTIONAL instruction. [ID_ISAR6.I8MM](#) indicates whether this instruction is supported in the T32 and A32 instruction sets.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

(FEAT_AA32I8MM)

31	30	29	28	27	26	25	24	23	22	21	20	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	0	N	1	M	0	Vm			
B												U																			

A1

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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	1	D	1	0	Vn				Vd				1	1	0	0	N	1	M	0	Vm			
B												U																			

T1

VUSMMLA{<q>}.S8 <Qd>, <Qn>, <Qm>

```
if InITBlock() then UNPREDICTABLE;
if !HaveAArch32Int8MatMulExt() 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;
integer d = UInt(D:Vd);
integer n = UInt(N:Vn);
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();
bits(128) operand1 = Q[n>>1];
bits(128) operand2 = Q[m>>1];
bits(128) addend   = Q[d>>1];

Q[d>>1] = MatMulAdd(addend, operand1, operand2, op1_unsigned, op2_unsigned);
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VUZP

Vector Unzip de-interleaves the elements of two vectors.

The elements of the vectors can be 8-bit, 16-bit, or 32-bit. There is no distinction between data types.

The following figure shows an example of the operation of VUZP doubleword operation for data type 8.

VUZP.8, doubleword

	Register state before operation								Register state after operation							
Dd	A7	A6	A5	A4	A3	A2	A1	A0	B6	B4	B2	B0	A6	A4	A2	A0
Dm	B7	B6	B5	B4	B3	B2	B1	B0	B7	B5	B3	B1	A7	A5	A3	A1

The following figure shows an example of the operation of VUZP quadword operation for data type 32.

VUZP.32, quadword

	Register state before operation				Register state after operation			
Qd	A3	A2	A1	A0	B2	B0	A2	A0
Qm	B3	B2	B1	B0	B3	B1	A3	A1

Depending on settings in the *CPACR*, *NSACR*, and *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0		Vd				0	0	0	1	0	Q	M	0		Vm		

64-bit SIMD vector (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); 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		

64-bit SIMD vector (Q == 0)

VUZP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VUZP{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); 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
            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
            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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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

64-bit SIMD vector

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

64-bit SIMD vector

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

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 *HCPTTR* registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see *Enabling Advanced SIMD and floating-point support*.

It has encodings from the following instruction sets: A32 ([A1](#)) and T32 ([T1](#)).

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	1	0	Vd	0	0	0	1	1	Q	M	0	Vm							

64-bit SIMD vector (Q == 0)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); 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							

64-bit SIMD vector (Q == 0)

VZIP{<c>}{<q>}.<dt> <Dd>, <Dm>

128-bit SIMD vector (Q == 1)

VZIP{<c>}{<q>}.<dt> <Qd>, <Qm>

```
if size == '11' || (Q == '0' && size == '10') then UNDEFINED;
if Q == '1' && (Vd<0> == '1' || Vm<0> == '1') then UNDEFINED;
quadword_operation = (Q == '1'); esize = 8 << UInt(size);
d = UInt(D:Vd); 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_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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 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

64-bit SIMD vector

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

64-bit SIMD vector

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.

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

Top-level encodings for A32

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				op0												op1															

Decode fields			Instruction details
cond	op0	op1	
!= 1111	00x		Data-processing and miscellaneous instructions
!= 1111	010		Load/Store Word, Unsigned Byte (immediate, literal)
!= 1111	011	0	Load/Store Word, Unsigned Byte (register)
!= 1111	011	1	Media instructions
	10x		Branch, branch with link, and block data transfer
	11x		System register access, Advanced SIMD, floating-point, and Supervisor call
1111	0xx		Unconditional instructions

Data-processing and miscellaneous instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	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	Extra load/store
0	0xxxx	1	00	1	Multiply and Accumulate
0	1xxxx	1	00	1	Synchronization primitives and Load-Acquire/Store-Release
0	10xx0	0			Miscellaneous
0	10xx0	1		0	Halfword Multiply and Accumulate
0	!= 10xx0			0	Data-processing register (immediate shift)
0	!= 10xx0	0		1	Data-processing register (register shift)
1					Data-processing immediate

Extra load/store

These instructions are under [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		Load/Store Dual, Half, Signed Byte (register)
1		Load/Store Dual, Half, Signed Byte (immediate, literal)

Load/Store Dual, Half, Signed Byte (register)

These instructions are under [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	STRH (register) — post-indexed
0	0	0	10	LDRD (register) — post-indexed
0	0	0	11	STRD (register) — post-indexed
0	0	1	01	LDRH (register) — post-indexed
0	0	1	10	LDRSB (register) — post-indexed
0	0	1	11	LDRSH (register) — post-indexed
0	1	0	01	STRHT
0	1	0	10	UNALLOCATED
0	1	0	11	UNALLOCATED
0	1	1	01	LDRHT
0	1	1	10	LDRSBT
0	1	1	11	LDRSHT
1		0	01	STRH (register) — pre-indexed
1		0	10	LDRD (register) — pre-indexed
1		0	11	STRD (register) — pre-indexed
1		1	01	LDRH (register) — pre-indexed
1		1	10	LDRSB (register) — pre-indexed
1		1	11	LDRSH (register) — pre-indexed

Load/Store Dual, Half, Signed Byte (immediate, literal)

These instructions are under [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	LDRD (literal)
!= 01	1	1111	01	LDRH (literal)
!= 01	1	1111	10	LDRSB (literal)
!= 01	1	1111	11	LDRSH (literal)
00	0	!= 1111	10	LDRD (immediate) — post-indexed
00	0		01	STRH (immediate) — post-indexed
00	0		11	STRD (immediate) — post-indexed
00	1	!= 1111	01	LDRH (immediate) — post-indexed
00	1	!= 1111	10	LDRSB (immediate) — post-indexed
00	1	!= 1111	11	LDRSH (immediate) — post-indexed
01	0	!= 1111	10	UNALLOCATED
01	0		01	STRHT
01	0		11	UNALLOCATED
01	1		01	LDRHT
01	1		10	LDRSBT
01	1		11	LDRSHT
10	0	!= 1111	10	LDRD (immediate) — offset
10	0		01	STRH (immediate) — offset

P:W	Decode fields			Instruction Details
	o1	Rn	op2	
10	0		11	STRD (immediate) – offset
10	1	!= 1111	01	LDRH (immediate) – offset
10	1	!= 1111	10	LDRSB (immediate) – offset
10	1	!= 1111	11	LDRSH (immediate) – offset
11	0	!= 1111	10	LDRD (immediate) – pre-indexed
11	0		01	STRH (immediate) – pre-indexed
11	0		11	STRD (immediate) – pre-indexed
11	1	!= 1111	01	LDRH (immediate) – pre-indexed
11	1	!= 1111	10	LDRSB (immediate) – pre-indexed
11	1	!= 1111	11	LDRSH (immediate) – pre-indexed

Multiply and Accumulate

These instructions are under [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		MUL, MULS
001		MLA, MLAS
010	0	UMAAL
010	1	UNALLOCATED
011	0	MLS
011	1	UNALLOCATED
100		UMULL, UMULLS
101		UMLAL, UMLALS
110		SMULL, SMULLS
111		SMLAL, SMLALS

Synchronization primitives and Load-Acquire/Store-Release

These instructions are under [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	Load/Store Exclusive and Load-Acquire/Store-Release

Load/Store Exclusive and Load-Acquire/Store-Release

These instructions are under [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	STL
00	0	0	1	UNALLOCATED
00	0	1	0	STLEX
00	0	1	1	STREX
00	1	0	0	LDA
00	1	0	1	UNALLOCATED
00	1	1	0	LDAEX
00	1	1	1	LDREX
01	0	0		UNALLOCATED
01	0	1	0	STLEXD
01	0	1	1	STREXD
01	1	0		UNALLOCATED
01	1	1	0	LDAEXD
01	1	1	1	LDREXD
10	0	0	0	STLB
10	0	0	1	UNALLOCATED
10	0	1	0	STLEXB
10	0	1	1	STREXB
10	1	0	0	LDAB
10	1	0	1	UNALLOCATED
10	1	1	0	LDAEXB
10	1	1	1	LDREXB
11	0	0	0	STLH
11	0	0	1	UNALLOCATED
11	0	1	0	STLEXH
11	0	1	1	STREXH
11	1	0	0	LDAH
11	1	0	1	UNALLOCATED
11	1	1	0	LDAEXH
11	1	1	1	LDREXH

Miscellaneous

These instructions are under [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	BX
01	010	BXJ
01	011	BLX (register)
01	110	UNALLOCATED
10	001	UNALLOCATED
10	010	UNALLOCATED
10	011	UNALLOCATED
10	110	UNALLOCATED
11	001	CLZ
11	010	UNALLOCATED
11	011	UNALLOCATED
11	110	ERET
	111	Exception Generation
	000	Move special register (register)
	100	Cyclic Redundancy Check
	101	Integer Saturating Arithmetic

Exception Generation

These instructions are under [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	HLT
01	BKPT
10	HVC
11	SMC

Move special register (register)

These instructions are under [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	MRS
x0	1	MRS (Banked register)
x1	0	MSR (register)
x1	1	MSR (Banked register)

Cyclic Redundancy Check

These instructions are under [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
sz	C	
00	0	CRC32 – CRC32B
00	1	CRC32C – CRC32CB
01	0	CRC32 – CRC32H
01	1	CRC32C – CRC32CH
10	0	CRC32 – CRC32W
10	1	CRC32C – CRC32CW
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

These instructions are under [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		QADD
01		QSUB
10		QDADD
11		QDSUB

Halfword Multiply and Accumulate

These instructions are under [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			SMLABB , SMLABT , SMLATB , SMLATT
01	0	0	SMLAWB , SMLAWT – SMLAWB
01	0	1	SMULWB , SMULWT – SMULWB

Decode fields			Instruction Details
opc	M	N	
01	1	0	SMLAWB, SMLAWT — SMLAWT
01	1	1	SMULWB, SMULWT — SMULWT
10			SMLALBB, SMLALBT, SMLALTB, SMLALTT
11			SMULBB, SMULBT, SMULTB, SMULTT

Data-processing register (immediate shift)

These instructions are under [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	
0x		Integer Data Processing (three register, immediate shift)
10	1	Integer Test and Compare (two register, immediate shift)
11		Logical Arithmetic (three register, immediate shift)

Integer Data Processing (three register, immediate shift)

These instructions are under [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	AND, ANDS (register) — shift or rotate by value
000			0000011	AND, ANDS (register) — rotate right with extend
001			!= 0000011	EOR, EORS (register) — shift or rotate by value
001			0000011	EOR, EORS (register) — rotate right with extend
010	0	!= 1101	!= 0000011	SUB, SUBS (register) — SUB, shift or rotate by value
010	0	!= 1101	0000011	SUB, SUBS (register) — SUB, rotate right with extend
010	0	1101	!= 0000011	SUB, SUBS (SP minus register) — SUB, shift or rotate by value
010	0	1101	0000011	SUB, SUBS (SP minus register) — SUB, rotate right with extend
010	1	!= 1101	!= 0000011	SUB, SUBS (register) — SUBS, shift or rotate by value
010	1	!= 1101	0000011	SUB, SUBS (register) — SUBS, rotate right with extend
010	1	1101	!= 0000011	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
010	1	1101	0000011	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
011			!= 0000011	RSB, RSBS (register) — shift or rotate by value
011			0000011	RSB, RSBS (register) — rotate right with extend
100	0	!= 1101	!= 0000011	ADD, ADDS (register) — ADD, shift or rotate by value
100	0	!= 1101	0000011	ADD, ADDS (register) — ADD, rotate right with extend
100	0	1101	!= 0000011	ADD, ADDS (SP plus register) — ADD, shift or rotate by value
100	0	1101	0000011	ADD, ADDS (SP plus register) — ADD, rotate right with extend

Decode fields				Instruction Details
opc	S	Rn	imm5:stype	
100	1	!= 1101	!= 0000011	ADD, ADDS (register) — ADDS, shift or rotate by value
100	1	!= 1101	0000011	ADD, ADDS (register) — ADDS, rotate right with extend
100	1	1101	!= 0000011	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
100	1	1101	0000011	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
101			!= 0000011	ADC, ADCS (register) — shift or rotate by value
101			0000011	ADC, ADCS (register) — rotate right with extend
110			!= 0000011	SBC, SBCS (register) — shift or rotate by value
110			0000011	SBC, SBCS (register) — rotate right with extend
111			!= 0000011	RSC, RSCS (register) — shift or rotate by value
111			0000011	RSC, RSCS (register) — rotate right with extend

Integer Test and Compare (two register, immediate shift)

These instructions are under [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:stype	
00	!= 0000011	TST (register) — shift or rotate by value
00	0000011	TST (register) — rotate right with extend
01	!= 0000011	TEQ (register) — shift or rotate by value
01	0000011	TEQ (register) — rotate right with extend
10	!= 0000011	CMP (register) — shift or rotate by value
10	0000011	CMP (register) — rotate right with extend
11	!= 0000011	CMN (register) — shift or rotate by value
11	0000011	CMN (register) — rotate right with extend

Logical Arithmetic (three register, immediate shift)

These instructions are under [Data-processing register \(immediate shift\)](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111	0	0	0	1	1	opc	S		Rn		Rd			imm5	stype	0		Rm													
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields		Instruction Details
opc	imm5:stype	
00	!= 0000011	ORR, ORRS (register) — shift or rotate by value
00	0000011	ORR, ORRS (register) — rotate right with extend
01	!= 0000011	MOV, MOVS (register) — shift or rotate by value
01	0000011	MOV, MOVS (register) — rotate right with extend
10	!= 0000011	BIC, BICS (register) — shift or rotate by value
10	0000011	BIC, BICS (register) — rotate right with extend
11	!= 0000011	MVN, MVNS (register) — shift or rotate by value

Decode fields		Instruction Details
opc	imm5:stype	
11	0000011	MVN, MVNS (register) – rotate right with extend

Data-processing register (register shift)

These instructions are under [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		Integer Data Processing (three register, register shift)
10	1	Integer Test and Compare (two register, register shift)
11		Logical Arithmetic (three register, register shift)

Integer Data Processing (three register, register shift)

These instructions are under [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		AND, ANDS (register-shifted register)
001		EOR, EORS (register-shifted register)
010		SUB, SUBS (register-shifted register)
011		RSB, RSBS (register-shifted register)
100		ADD, ADDS (register-shifted register)
101		ADC, ADCS (register-shifted register)
110		SBC, SBCS (register-shifted register)
111		RSC, RSCS (register-shifted register)

Integer Test and Compare (two register, register shift)

These instructions are under [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		TST (register-shifted register)
01		TEQ (register-shifted register)

Decode fields
opc **Instruction Details**

10	CMP (register-shifted register)
11	CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

These instructions are under [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
opc **Instruction Details**

00	ORR, ORRS (register-shifted register)
01	MOV, MOVS (register-shifted register)
10	BIC, BICS (register-shifted register)
11	MVN, MVNS (register-shifted register)

Data-processing immediate

These instructions are under [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
op0 **op1** **Instruction details**

0x		Integer Data Processing (two register and immediate)
10	00	Move Halfword (immediate)
10	10	Move Special Register and Hints (immediate)
10	x1	Integer Test and Compare (one register and immediate)
11		Logical Arithmetic (two register and immediate)

Integer Data Processing (two register and immediate)

These instructions are under [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
opc **S** **Rn** **Instruction Details**

000			AND, ANDS (immediate)
001			EOR, EORS (immediate)
010	0	!= 11x1	SUB, SUBS (immediate) – SUB
010	0	1101	SUB, SUBS (SP minus immediate) – SUB
010	0	1111	ADR – A2

Decode fields			Instruction Details
opc	S	Rn	
010	1	!= 1101	SUB, SUBS (immediate) – SUBS
010	1	1101	SUB, SUBS (SP minus immediate) – SUBS
011			RSB, RSBS (immediate)
100	0	!= 11x1	ADD, ADDS (immediate) – ADD
100	0	1101	ADD, ADDS (SP plus immediate) – ADD
100	0	1111	ADR – A1
100	1	!= 1101	ADD, ADDS (immediate) – ADDS
100	1	1101	ADD, ADDS (SP plus immediate) – ADDS
101			ADC, ADCS (immediate)
110			SBC, SBCS (immediate)
111			RSC, RSCS (immediate)

Move Halfword (immediate)

These instructions are under [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		MOV, MOVS (immediate)
1		MOVT

Move Special Register and Hints (immediate)

These instructions are under [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		MSR (immediate)	-
00000	xxxx00000000	NOP	-
00000	xxxx00000001	YIELD	-
00000	xxxx00000010	WFE	-
00000	xxxx00000011	WFI	-
00000	xxxx00000100	SEV	-
00000	xxxx00000101	SEVL	-
00000	xxxx0000011x	Reserved hint, behaves as NOP	-
00000	xxxx00001xxx	Reserved hint, behaves as NOP	-
00000	xxxx00010000	ESB	FEAT_RAS
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	TSB CSYNC	FEAT_TRF

Decode fields		Instruction Details	Feature
R:imm4	imm12		
00000	XXXX00010011	Reserved hint, behaves as NOP	-
00000	XXXX00010100	CSDB	-
00000	XXXX00010101	Reserved hint, behaves as NOP	-
00000	XXXX0001011x	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	DBG	-

Integer Test and Compare (one register and immediate)

These instructions are under [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	Instruction Details
opc	
00	TST (immediate)
01	TEQ (immediate)
10	CMP (immediate)
11	CMN (immediate)

Logical Arithmetic (two register and immediate)

These instructions are under [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	Instruction Details
opc	
00	ORR, ORRS (immediate)
01	MOV, MOVS (immediate)
10	BIC, BICS (immediate)
11	MVN, MVNS (immediate)

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

P:W	Decode fields			Rn	Instruction Details
	o2	o1			
!= 01	0	1		1111	LDR (literal)
!= 01	1	1		1111	LDRB (literal)
00	0	0			STR (immediate) — post-indexed
00	0	1	!= 1111		LDR (immediate) — post-indexed
00	1	0			STRB (immediate) — post-indexed
00	1	1	!= 1111		LDRB (immediate) — post-indexed
01	0	0			STRT
01	0	1			LDRT
01	1	0			STRBT
01	1	1			LDRBT
10	0	0			STR (immediate) — offset
10	0	1	!= 1111		LDR (immediate) — offset
10	1	0			STRB (immediate) — offset
10	1	1	!= 1111		LDRB (immediate) — offset
11	0	0			STR (immediate) — pre-indexed
11	0	1	!= 1111		LDR (immediate) — pre-indexed
11	1	0			STRB (immediate) — pre-indexed
11	1	1	!= 1111		LDRB (immediate) — pre-indexed

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

P	Decode fields			Instruction Details
	o2	W	o1	
0	0	0	0	STR (register) — post-indexed
0	0	0	1	LDR (register) — post-indexed
0	0	1	0	STRT
0	0	1	1	LDRT
0	1	0	0	STRB (register) — post-indexed
0	1	0	1	LDRB (register) — post-indexed
0	1	1	0	STRBT
0	1	1	1	LDRBT
1	0		0	STR (register) — pre-indexed
1	0		1	LDR (register) — pre-indexed
1	1		0	STRB (register) — pre-indexed
1	1		1	LDRB (register) — pre-indexed

Media instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	23	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		Parallel Arithmetic
01000	101	SEL
01000	001	UNALLOCATED
01000	xx0	PKHBT, PKHTB
01001	x01	UNALLOCATED
01001	xx0	UNALLOCATED
0110x	x01	UNALLOCATED
0110x	xx0	UNALLOCATED
01x10	001	Saturate 16-bit
01x10	101	UNALLOCATED
01x11	x01	Reverse Bit/Byte
01x1x	xx0	Saturate 32-bit
01xxx	111	UNALLOCATED
01xxx	011	Extend and Add
10xxx		Signed multiply, Divide
11000	000	Unsigned Sum of Absolute Differences
11000	100	UNALLOCATED
11001	x00	UNALLOCATED
1101x	x00	UNALLOCATED
110xx	111	UNALLOCATED
1110x	111	UNALLOCATED
1110x	x00	Bitfield Insert
11110	111	UNALLOCATED
11111	111	Permanently UNDEFINED
1111x	x00	UNALLOCATED
11x0x	x10	UNALLOCATED
11x1x	x10	Bitfield Extract
11xxx	011	UNALLOCATED
11xxx	x01	UNALLOCATED

Parallel Arithmetic

These instructions are under [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	SADD16
001	0	01	SASX
001	0	10	SSAX
001	0	11	SSUB16
001	1	00	SADD8
001	1	01	UNALLOCATED

Decode fields			Instruction Details
op1	B	op2	
001	1	10	UNALLOCATED
001	1	11	SSUB8
010	0	00	QADD16
010	0	01	QASX
010	0	10	QSAX
010	0	11	QSUB16
010	1	00	QADD8
010	1	01	UNALLOCATED
010	1	10	UNALLOCATED
010	1	11	QSUB8
011	0	00	SHADD16
011	0	01	SHASX
011	0	10	SHSAX
011	0	11	SHSUB16
011	1	00	SHADD8
011	1	01	UNALLOCATED
011	1	10	UNALLOCATED
011	1	11	SHSUB8
100			UNALLOCATED
101	0	00	UADD16
101	0	01	UASX
101	0	10	USAX
101	0	11	USUB16
101	1	00	UADD8
101	1	01	UNALLOCATED
101	1	10	UNALLOCATED
101	1	11	USUB8
110	0	00	UQADD16
110	0	01	UQASX
110	0	10	UQSAX
110	0	11	UQSUB16
110	1	00	UQADD8
110	1	01	UNALLOCATED
110	1	10	UNALLOCATED
110	1	11	UQSUB8
111	0	00	UHADD16
111	0	01	UHASX
111	0	10	UHSAX
111	0	11	UHSUB16
111	1	00	UHADD8
111	1	01	UNALLOCATED
111	1	10	UNALLOCATED
111	1	11	UHSUB8

Saturate 16-bit

These instructions are under [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	SSAT16
1	USAT16

Reverse Bit/Byte

These instructions are under [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	REV
0	1	REV16
1	0	RBIT
1	1	REVSH

Saturate 32-bit

These instructions are under [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	SSAT
1	USAT

Extend and Add

These instructions are under [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
		SXTAB16

Decode fields			Instruction Details
U	op	Rn	
0	00	1111	SXTB16
0	10	!= 1111	SXTAB
0	10	1111	SXTB
0	11	!= 1111	SXTAH
0	11	1111	SXTH
1	00	!= 1111	UXTAB16
1	00	1111	UXTB16
1	10	!= 1111	UXTAB
1	10	1111	UXTB
1	11	!= 1111	UXTAH
1	11	1111	UXTH

Signed multiply, Divide

These instructions are under [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	SMLAD, SMLADX – SMLAD
000	!= 1111	001	SMLAD, SMLADX – SMLADX
000	!= 1111	010	SMLSD, SMLSDX – SMLSD
000	!= 1111	011	SMLSD, SMLSDX – SMLSDX
000		1xx	UNALLOCATED
000	1111	000	SMUAD, SMUADX – SMUAD
000	1111	001	SMUAD, SMUADX – SMUADX
000	1111	010	SMUSD, SMUSDX – SMUSD
000	1111	011	SMUSD, SMUSDX – SMUSDX
001		000	SDIV
001		!= 000	UNALLOCATED
010			UNALLOCATED
011		000	UDIV
011		!= 000	UNALLOCATED
100		000	SMLALD, SMLALDX – SMLALD
100		001	SMLALD, SMLALDX – SMLALDX
100		010	SMLSLD, SMLSLDX – SMLSLD
100		011	SMLSLD, SMLSLDX – SMLSLDX
100		1xx	UNALLOCATED
101	!= 1111	000	SMMLA, SMMLAR – SMMLA
101	!= 1111	001	SMMLA, SMMLAR – SMMLAR
101		01x	UNALLOCATED
101		10x	UNALLOCATED
101		110	SMMLS, SMMLSR – SMMLS
101		111	SMMLS, SMMLSR – SMMLSR

Decode fields			Instruction Details
op1	Ra	op2	
101	1111	000	SMMUL, SMMULR – SMMUL
101	1111	001	SMMUL, SMMULR – SMMULR
11x			UNALLOCATED

Unsigned Sum of Absolute Differences

These instructions are under [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																			Rn
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Ra	
!= 1111	USADA8
1111	USAD8

Bitfield Insert

These instructions are under [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																			Rn
cond																																

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Rn	
!= 1111	BFI
1111	BFC

Permanently UNDEFINED

These instructions are under [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																									imm4
												imm12																					
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	UDF

Bitfield Extract

These instructions are under [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	SBFX
1	UBFX

Branch, branch with link, and block data transfer

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	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	Exception Save/Restore
!= 1111	0	Load/Store Multiple
	1	Branch (immediate)

Exception Save/Restore

These instructions are under [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	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement After
0	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement After
0	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment After
0	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment After
1	0	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Decrement Before
1	0	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Decrement Before
		1	1	UNALLOCATED
1	1	0	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB — Increment Before
1	1	1	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB — Increment Before

Load/Store Multiple

These instructions are under [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		STMDA, STMED
0	0	0	1		LMDA, LDMFA
0	1	0	0		STM, STMIA, STMEA
0	1	0	1		LDM, LDMIA, LDMFD
		1	0		STM (User registers)
1	0	0	0		STMDB, STMFD
1	0	0	1		LDMDB, LDMEA
		1	1	0XXXXXXXXXXXXXXXXXX	LDM (User registers)
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED
		1	1	1XXXXXXXXXXXXXXXXXX	LDM (exception return)

Branch (immediate)

These instructions are under [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		B
!= 1111	1		BL, BLX (immediate) – A1
1111			BL, BLX (immediate) – A2

System register access, Advanced SIMD, floating-point, and Supervisor call

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	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			Supervisor call
1111	!= 11	1x		Unconditional Advanced SIMD and floating-point instructions
!= 1111	0x	1x		Advanced SIMD and System register load/store and 64-bit move
!= 1111	10	1x	1	Advanced SIMD and System register 32-bit move
!= 1111	10	10	0	Floating-point data-processing
!= 1111	10	11	0	UNALLOCATED

Supervisor call

These instructions are under [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	SVC

Unconditional Advanced SIMD and floating-point instructions

These instructions are under [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			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select
101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

These instructions are under [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 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	VCADD — 64-bit SIMD vector	FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 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	VMMLA	FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	FEAT_AA32BF16
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	FEAT_FHM
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA	FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	FEAT_DotProd
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	FEAT_AA32BF16
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	FEAT_FHM
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 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	VCMLA	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

These instructions are under [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 size	Instruction Details	Feature
01	VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10	VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11	VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

These instructions are under [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 size	op	Instruction Details	Feature
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

These instructions are under [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 size	op	Instruction Details	Feature
01		UNALLOCATED	-
10	0	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

Floating-point directed convert to integer

These instructions are under [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

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	RM	size	op		
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-
1	11	11		VCVTM (floating-point) — double-precision scalar	-

Advanced SIMD and floating-point multiply with accumulate

These instructions are under [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	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	FEAT_FCMA
0	00		1	VFMAL (by scalar)	FEAT_FHM
0	01		1	VFMSL (by scalar)	FEAT_FHM
0	10		1	UNALLOCATED	-
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	FEAT_FCMA
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	FEAT_FCMA

Advanced SIMD and floating-point dot product

These instructions are under [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	VDOT (by element) — 64-bit SIMD vector	FEAT_AA32BF16

Decode fields					Instruction Details	Feature
op1	op2	op4	Q	U		
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	FEAT_AA32BF16
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	FEAT_DotProd
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	FEAT_AA32I8MM
1	01	1			UNALLOCATED	-
1	1x	1			UNALLOCATED	-

Advanced SIMD and System register load/store and 64-bit move

These instructions are under [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	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register load/store
	10	UNALLOCATED

Advanced SIMD and floating-point 64-bit move

These instructions are under [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	VMOV (between two general-purpose registers and two single-precision registers) — from general-purpose registers

Decode fields					Instruction Details
D	op	size	opc2	o3	
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) — to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) — to general-purpose registers

System register 64-bit move

These instructions are under [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	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

These instructions are under [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											Instruction Details	Feature	
P	U	W	L	Rn	size	imm8							
0	0	1										UNALLOCATED	-
0	1				0x							UNALLOCATED	-
0	1		0		10							VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	xxxxxxx0						VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	xxxxxxx1						FSTMDBX, FSTMIAx — Increment After	-
0	1		1		10							VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	xxxxxxx0						VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	xxxxxxx1						FLDM*x (FLDMDBX, FLDMIAX) — Increment After	-
1		0	0		01							VSTR — half-precision scalar	FEAT_FP16
1		0	0		10							VSTR — single-precision scalar	-
1		0	0		11							VSTR — double-precision scalar	-

P	U	W	Decode fields			imm8	Instruction Details	Feature
			L	Rn	size			
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	xxxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register load/store

These instructions are under [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

P:U:W	D	L	Decode fields			cp15	Instruction Details
			Rn	CRd			
!= 000	0			!= 0101	0	UNALLOCATED	
!= 000	0	1	1111	0101	0	LDC (literal)	
!= 000					1	UNALLOCATED	
!= 000	1			0101	0	UNALLOCATED	
0x1	0	0		0101	0	STC — post-indexed	
0x1	0	1	!= 1111	0101	0	LDC (immediate) — post-indexed	
010	0	0		0101	0	STC — unindexed	
010	0	1	!= 1111	0101	0	LDC (immediate) — unindexed	
1x0	0	0		0101	0	STC — offset	
1x0	0	1	!= 1111	0101	0	LDC (immediate) — offset	
1x1	0	0		0101	0	STC — pre-indexed	
1x1	0	1	!= 1111	0101	0	LDC (immediate) — pre-indexed	

Advanced SIMD and System register 32-bit move

These instructions are under [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	Architecture version
op0	op1		
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under [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	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under [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		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

These instructions are under [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 L	Instruction Details
0	MCR
1	MRC

Floating-point data-processing

These instructions are under [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 op0	op1	Instruction details
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under [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	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under [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		Instruction Details	Feature
size			
00		UNALLOCATED	-
01		VMOV (immediate) — half-precision scalar	FEAT_FP16
10		VMOV (immediate) — single-precision scalar	-
11		VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under [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	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Unconditional instructions

These instructions are under the [top-level](#).

31	30	29	28	27	26	25	24	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		Miscellaneous
01x		Advanced SIMD data-processing
1xx	1	Memory hints and barriers
100	0	Advanced SIMD element or structure load/store
101	0	UNALLOCATED
11x	0	UNALLOCATED

Miscellaneous

These instructions are under [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	Architecture version
op0	op1		
0XXXX		UNALLOCATED	-
10000	xx0x	Change Process State	-
10001	1000	UNALLOCATED	-
10001	x100	UNALLOCATED	-
10001	xx01	UNALLOCATED	-
10001	0000	SETPAN	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

These instructions are under [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	SETEND
00	1	0				CPS, CPSID, CPSIE — change mode
10		0				CPS, CPSID, CPSIE — interrupt enable and change mode
		1	0	0	1XXXX	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED
11		0				CPS, CPSID, CPSIE — interrupt disable and change mode

Advanced SIMD data-processing

These instructions are under [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		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

These instructions are under [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	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) – A2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) – A1	-
		0011		1	VCGE (register) – A1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-

U	Decode fields			o1	Instruction Details	Feature
	size	opc	Q			
1	0x	1110		0	VCGE (register) – A2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) – A2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) – A1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

These instructions are under [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				VEXT (byte elements)
1	11	0x			Advanced SIMD two registers misc
1	11	10			VTBL, VTBX
1	11	11			Advanced SIMD duplicate (scalar)
	!= 11			0	Advanced SIMD three registers of different lengths
	!= 11			1	Advanced SIMD two registers and a scalar

Advanced SIMD two registers misc

These instructions are under [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		VREV64	-
	00	0001		VREV32	-
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	-
	00	0110	1	AESD	-
	00	0111	0	AESMC	-
	00	0111	1	AESIMC	-
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN — VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN — VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) — single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) — half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRTE	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

These instructions are under [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	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

Advanced SIMD three registers of different lengths

These instructions are under [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

U	Decode fields opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)

Decode fields U	opc	Instruction Details
1	1001	UNALLOCATED
1	1011	UNALLOCATED
1	1101	UNALLOCATED
	1111	UNALLOCATED

Advanced SIMD two registers and a scalar

These instructions are under [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	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQDMLAH	-
1	1011	UNALLOCATED	-
	1110	VQDMLAH	FEAT_RDM
	1111	VQDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

These instructions are under [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 op0	Instruction details
000XXXXXXXXXXXX0	Advanced SIMD one register and modified immediate
!= 000XXXXXXXXXXXX0	Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under [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	VMOV (immediate) – A1
0xx0	1	VMVN (immediate) – A1
0xx1	0	VORR (immediate) – A1
0xx1	1	VBIC (immediate) – A1
10x0	0	VMOV (immediate) – A3
10x0	1	VMVN (immediate) – A2
10x1	0	VORR (immediate) – A2
10x1	1	VBIC (immediate) – A2
11xx	0	VMOV (immediate) – A4
110x	1	VMVN (immediate) – A3
1110	1	VMOV (immediate) – A5
1111	1	UNALLOCATED

Advanced SIMD two registers and shift amount

These instructions are under [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		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSRHR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) – VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN – VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN – VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) – VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN – VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN – VQRSHRUN

Memory hints and barriers

These instructions are under [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		Barriers
011x1		CONSTRAINED UNPREDICTABLE
0xxx0		Preload (immediate)
1xxx0	0	Preload (register)
1xxx1	0	CONSTRAINED UNPREDICTABLE
1xxxx	1	UNALLOCATED

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Barriers

These instructions are under [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		CLREX	-
001x		CONSTRAINED UNPREDICTABLE	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101		DMB	-
0110		ISB	-
0111		SB	FEAT_SB
1xxx		CONSTRAINED UNPREDICTABLE	-

The behavior of the CONSTRAINED UNPREDICTABLE encodings in this table is described in [CONSTRAINED UNPREDICTABLE behavior for A32 and T32 instruction encodings](#)

Preload (immediate)

These instructions are under [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	0	D	U	R	0	1	Rn			(1)	(1)	(1)	(1)	imm12												

Decode fields			Instruction Details
D	R	Rn	
0	0		Reserved hint, behaves as NOP
0	1		PLI (immediate, literal)
1		1111	PLD (literal)
1	0	!= 1111	PLD, PLDW (immediate) — preload write
1	1	!= 1111	PLD, PLDW (immediate) — preload read

Preload (register)

These instructions are under [Memory hints and barriers](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	1	1	D	U	o2	0	1		Rn		(1)	(1)	(1)	(1)		imm5		stype	0							Rm	

Decode fields			Instruction Details
D	o2	imm5:stype	
0	0		Reserved hint, behaves as NOP
0	1	!= 0000011	PLI (register) – shift or rotate by value
0	1	0000011	PLI (register) – rotate right with extend
1	0	!= 0000011	PLD, PLDW (register) – preload write, optional shift or rotate
1	0	0000011	PLD, PLDW (register) – preload write, rotate right with extend
1	1	!= 0000011	PLD, PLDW (register) – preload read, optional shift or rotate
1	1	0000011	PLD, PLDW (register) – preload read, rotate right with extend

Advanced SIMD element or structure load/store

These instructions are under [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		Advanced SIMD load/store multiple structures
1	11	Advanced SIMD load single structure to all lanes
1	!= 11	Advanced SIMD load/store single structure to one lane

Advanced SIMD load/store multiple structures

These instructions are under [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	VST4 (multiple 4-element structures)
0	000x	1101	VST4 (multiple 4-element structures)
0	000x	1111	VST4 (multiple 4-element structures)
0	0010	!= 11x1	VST1 (multiple single elements)
0	0010	1101	VST1 (multiple single elements)
0	0010	1111	VST1 (multiple single elements)
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)

L	Decode fields		Instruction Details
	itype	Rm	
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

These instructions are under [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			

L	Decode fields		Instruction Details
	N	a	
0			UNALLOCATED
1	00		!= 11x1 VLD1 (single element to all lanes)
1	00		1101 VLD1 (single element to all lanes)
1	00		1111 VLD1 (single element to all lanes)

Decode fields				Instruction Details
L	N	a	Rm	
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under [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	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)

L	Decode fields			Instruction Details
	size	N	Rm	
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Top-level encodings for T32

15	14	13	12	11	10	9	8	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		16-bit
111	00	B – T2
111	!= 00	32-bit

16-bit

These instructions are under the [top-level](#).

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	Shift (immediate), add, subtract, move, and compare
010000	Data-processing (two low registers)
010001	Special data instructions and branch and exchange
01001x	LDR (literal) – T1
0101xx	Load/store (register offset)
011xxx	Load/store word/byte (immediate offset)
1000xx	Load/store halfword (immediate offset)
1001xx	Load/store (SP-relative)
1010xx	Add PC/SP (immediate)
1011xx	Miscellaneous 16-bit instructions
1100xx	Load/store multiple
1101xx	Conditional branch, and Supervisor Call

Shift (immediate), add, subtract, move, and compare

These instructions are under [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	Add, subtract (three low registers)
0	11	1	Add, subtract (two low registers and immediate)
0	!= 11		MOV, MOVS (register) – T2
1			Add, subtract, compare, move (one low register and immediate)

Add, subtract (three low registers)

These instructions are under [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	ADD, ADDS (register)
1	SUB, SUBS (register)

Add, subtract (two low registers and immediate)

These instructions are under [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	ADD, ADDS (immediate)
1	SUB, SUBS (immediate)

Add, subtract, compare, move (one low register and immediate)

These instructions are under [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	MOV, MOVS (immediate)
01	CMP (immediate)
10	ADD, ADDS (immediate)
11	SUB, SUBS (immediate)

Data-processing (two low registers)

These instructions are under [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	AND, ANDS (register)
0001	EOR, EORS (register)
0010	MOV, MOVS (register-shifted register) — logical shift left
0011	MOV, MOVS (register-shifted register) — logical shift right
0100	MOV, MOVS (register-shifted register) — arithmetic shift right
0101	ADC, ADCS (register)
0110	SBC, SBCS (register)
0111	MOV, MOVS (register-shifted register) — rotate right
1000	TST (register)
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)

Decode fields op	Instruction Details
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Special data instructions and branch and exchange

These instructions are under [16-bit](#).

15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
010001	op0

Decode fields op0	Instruction details
11	Branch and exchange
!= 11	Add, subtract, compare, move (two high registers)

Branch and exchange

These instructions are under [Special data instructions and branch and exchange](#).

15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1 1 1	L Rm (0)(0)(0)

Decode fields L	Instruction Details
0	BX
1	BLX (register)

Add, subtract, compare, move (two high registers)

These instructions are under [Special data instructions and branch and exchange](#).

15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
0 1 0 0 0 1	!= 11 D Rs Rd
op	

The following constraints also apply to this encoding: op != 11 && op != 11

op	Decode fields D:Rd Rs		Instruction Details
00	!= 1101	!= 1101	ADD, ADDS (register)
00		1101	ADD, ADDS (SP plus register) – T1
00	1101	!= 1101	ADD, ADDS (SP plus register) – T2
01			CMP (register)
10			MOV, MOVS (register)

Load/store (register offset)

These instructions are under [16-bit](#).

15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
0 1 0 1	L B H Rm Rn Rt

Decode fields			Instruction Details
L	B	H	
0	0	0	STR (register)
0	0	1	STRH (register)
0	1	0	STRB (register)
0	1	1	LDRSB (register)
1	0	0	LDR (register)
1	0	1	LDRH (register)
1	1	0	LDRB (register)
1	1	1	LDRSH (register)

Load/store word/byte (immediate offset)

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	1	B	L	imm5					Rn		Rt			

Decode fields		Instruction Details
B	L	
0	0	STR (immediate)
0	1	LDR (immediate)
1	0	STRB (immediate)
1	1	LDRB (immediate)

Load/store halfword (immediate offset)

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	L	imm5					Rn		Rt			

Decode fields		Instruction Details
L		
0		STRH (immediate)
1		LDRH (immediate)

Load/store (SP-relative)

These instructions are under [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		STR (immediate)
1		LDR (immediate)

Add PC/SP (immediate)

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	SP	Rd		imm8								

Decode fields SP	Instruction Details
0	ADR
1	ADD, ADDS (SP plus immediate)

Miscellaneous 16-bit instructions

These instructions are under [16-bit](#).

15141312111098 7 6 5 43210
1011 op0 op1 op2 op3

op0	Decode fields op1	op2	op3	Instruction details	Architecture version
0000				Adjust SP (immediate)	-
0010				Extend	-
0110	00	0		SETPAN	FEAT_PAN
0110	00	1		UNALLOCATED	-
0110	01			Change Processor State	-
0110	1x			UNALLOCATED	-
0111				UNALLOCATED	-
1000				UNALLOCATED	-
1010	10			HLT	-
1010	!= 10			Reverse bytes	-
1110				BKPT	-
1111			0000	Hints	-
1111			!= 0000	IT	-
x0x1				CBNZ, CBZ	-
x10x				Push and Pop	-

Adjust SP (immediate)

These instructions are under [Miscellaneous 16-bit instructions](#).

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
1 0 1 1 0 0 0 0 S imm7

Decode fields S	Instruction Details
0	ADD, ADDS (SP plus immediate)
1	SUB, SUBS (SP minus immediate)

Extend

These instructions are under [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 U	Decode fields B	Instruction Details
0	0	SXTH
0	1	SXTB
1	0	UXTH

Decode fields		Instruction Details
U	B	
1	1	UXTB

Change Processor State

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	0	0	1	op	flags				

Decode fields		Instruction Details
op	flags	
0		SETEND
1	0XXXX	CPS, CPSID, CPSIE – interrupt enable
1	1XXXX	CPS, CPSID, CPSIE – interrupt disable

Reverse bytes

These instructions are under [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		REV
01		REV16
11		REVSH

Hints

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	hint	0 0 0 0						

Decode fields		Instruction Details
hint		
0000		NOP
0001		YIELD
0010		WFE
0011		WFI
0100		SEV
0101		SEVL
011X		Reserved hint, behaves as NOP
1XXX		Reserved hint, behaves as NOP

Push and Pop

These instructions are under [Miscellaneous 16-bit instructions](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	L	1	0	P	register_list							

Decode fields	Instruction Details
L	
0	PUSH
1	POP

Load/store multiple

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	L	Rn	register_list									

Decode fields	Instruction Details
L	
0	STM, STMIA, STMEA
1	LDM, LDMIA, LDMFD

Conditional branch, and Supervisor Call

These instructions are under [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	Exception generation
!= 111x	B – T1

Exception generation

These instructions are under [Conditional branch, and Supervisor Call](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	1	1	1	1	S	imm8							

Decode fields	Instruction Details
S	
0	UDF
1	SVC

32-bit

These instructions are under the [top-level](#).

15	14	13	12	11	10	9	8	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			System register access, Advanced SIMD, and floating-point

0100	xx0xx		Load/store multiple
0100	xx1xx		Load/store dual, load/store exclusive, load-acquire/store-release, and table branch
0101			Data-processing (shifted register)
10xx		1	Branches and miscellaneous control
10x0		0	Data-processing (modified immediate)
10x1	xxxx0	0	Data-processing (plain binary immediate)
10x1	xxxx1	0	UNALLOCATED
1100	1xxx0		Advanced SIMD element or structure load/store
1100	!= 1xxx0		Load/store single
1101	0xxxx		Data-processing (register)
1101	10xxx		Multiply, multiply accumulate, and absolute difference
1101	11xxx		Long multiply and divide

System register access, Advanced SIMD, and floating-point

These instructions are under [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			Advanced SIMD data-processing
0	0x	1x		Advanced SIMD and System register load/store and 64-bit move
0	10	1x	1	Advanced SIMD and System register 32-bit move
0	10	10	0	Floating-point data-processing
0	10	11	0	UNALLOCATED
1	!= 11	1x		Additional Advanced SIMD and floating-point instructions

Advanced SIMD data-processing

These instructions are under [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		Advanced SIMD three registers of the same length
1	0	Advanced SIMD two registers, or three registers of different lengths
1	1	Advanced SIMD shifts and immediate generation

Advanced SIMD three registers of the same length

These instructions are under [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			o1	Instruction Details	Feature
	size	opc	Q			
0	0x	1100		1	VFMA	-
0	0x	1101		0	VADD (floating-point)	-
0	0x	1101		1	VMLA (floating-point)	-
0	0x	1110		0	VCEQ (register) – T2	-
0	0x	1111		0	VMAX (floating-point)	-
0	0x	1111		1	VRECPS	-
		0000		0	VHADD	-
0	00	0001		1	VAND (register)	-
		0000		1	VQADD	-
		0001		0	VRHADD	-
0	00	1100		0	SHA1C	-
		0010		0	VHSUB	-
0	01	0001		1	VBIC (register)	-
		0010		1	VQSUB	-
		0011		0	VCGT (register) – T1	-
		0011		1	VCGE (register) – T1	-
0	01	1100		0	SHA1P	-
0	1x	1100		1	VFMS	-
0	1x	1101		0	VSUB (floating-point)	-
0	1x	1101		1	VMLS (floating-point)	-
0	1x	1110		0	UNALLOCATED	-
0	1x	1111		0	VMIN (floating-point)	-
0	1x	1111		1	VRSQRTS	-
		0100		0	VSHL (register)	-
0		1000		0	VADD (integer)	-
0	10	0001		1	VORR (register)	-
0		1000		1	VTST	-
		0100		1	VQSHL (register)	-
0		1001		0	VMLA (integer)	-
		0101		0	VRSHL	-
		0101		1	VQRSHL	-
0		1011		0	VQDMULH	-
0	10	1100		0	SHA1M	-
0		1011		1	VPADD (integer)	-
		0110		0	VMAX (integer)	-
0	11	0001		1	VORN (register)	-
		0110		1	VMIN (integer)	-
		0111		0	VABD (integer)	-
		0111		1	VABA	-
0	11	1100		0	SHA1SU0	-
1	0x	1101		0	VPADD (floating-point)	-
1	0x	1101		1	VMUL (floating-point)	-
1	0x	1110		0	VCGE (register) – T2	-
1	0x	1110		1	VACGE	-
1	0x	1111	0	0	VPMAX (floating-point)	-
1	0x	1111		1	VMAXNM	-
1	00	0001		1	VEOR	-

U	Decode fields			o1	Instruction Details	Feature
	size	opc	Q			
		1001		1	VMUL (integer and polynomial)	-
1	00	1100		0	SHA256H	-
		1010	0	0	VPMAX (integer)	-
1	01	0001		1	VBSL	-
		1010	0	1	VPMIN (integer)	-
		1010	1		UNALLOCATED	-
1	01	1100		0	SHA256H2	-
1	1x	1101		0	VABD (floating-point)	-
1	1x	1110		0	VCGT (register) – T2	-
1	1x	1110		1	VACGT	-
1	1x	1111	0	0	VPMIN (floating-point)	-
1	1x	1111		1	VMINNM	-
1		1000		0	VSUB (integer)	-
1	10	0001		1	VBIT	-
1		1000		1	VCEQ (register) – T1	-
1		1001		0	VMLS (integer)	-
1		1011		0	VQRDMULH	-
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	FEAT_RDM
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	FEAT_RDM
1		1111	1	0	UNALLOCATED	-

Advanced SIMD two registers, or three registers of different lengths

These instructions are under [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														

op0	Decode fields			Instruction details
	op1	op2	op3	
0	11			VEXT (byte elements)
1	11	0x		Advanced SIMD two registers misc
1	11	10		VTBL, VTBX
1	11	11		Advanced SIMD duplicate (scalar)
	!= 11		0	Advanced SIMD three registers of different lengths
	!= 11		1	Advanced SIMD two registers and a scalar

Advanced SIMD two registers misc

These instructions are under [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			Instruction Details	Feature
	opc1	opc2	Q		
	00	0000		VREV64	-
	00	0001		VREV32	-

size	Decode fields		Q	Instruction Details	Feature
	opc1	opc2			
	00	0010		VREV16	-
	00	0011		UNALLOCATED	-
	00	010x		VPADDL	-
	00	0110	0	AESE	-
	00	0110	1	AESD	-
	00	0111	0	AESMC	-
	00	0111	1	AESIMC	-
	00	1000		VCLS	-
00	10	0000		VSWP	-
	00	1001		VCLZ	-
	00	1010		VCNT	-
	00	1011		VMVN (register)	-
00	10	1100	1	UNALLOCATED	-
	00	110x		VPADAL	-
	00	1110		VQABS	-
	00	1111		VQNEG	-
	01	x000		VCGT (immediate #0)	-
	01	x001		VCGE (immediate #0)	-
	01	x010		VCEQ (immediate #0)	-
	01	x011		VCLE (immediate #0)	-
	01	x100		VCLT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	FEAT_AA32BF16
	10	0001		VTRN	-
	10	0010		VUZP	-
	10	0011		VZIP	-
	10	0100	0	VMOVN	-
	10	0100	1	VQMOVN, VQMOVUN – VQMOVUN	-
	10	0101		VQMOVN, VQMOVUN – VQMOVN	-
	10	0110	0	VSHLL	-
	10	0111	0	SHA1SU1	-
	10	0111	1	SHA256SU0	-
	10	1000		VRINTN (Advanced SIMD)	-
	10	1001		VRINTX (Advanced SIMD)	-
	10	1010		VRINTA (Advanced SIMD)	-
	10	1011		VRINTZ (Advanced SIMD)	-
10	10	1100	1	UNALLOCATED	-
	10	1100	0	VCVT (between half-precision and single-precision, Advanced SIMD) – single-precision to half-precision	-
	10	1101		VRINTM (Advanced SIMD)	-
	10	1110	0	VCVT (between half-precision and single-precision, Advanced SIMD) – half-precision to single-precision	-
	10	1110	1	UNALLOCATED	-
	10	1111		VRINTP (Advanced SIMD)	-
	11	000x		VCVTA (Advanced SIMD)	-
	11	001x		VCVTN (Advanced SIMD)	-

Decode fields size	opc1	opc2	Q	Instruction Details	Feature
	11	010x		VCVTP (Advanced SIMD)	-
	11	011x		VCVTM (Advanced SIMD)	-
	11	10x0		VRECPE	-
	11	10x1		VRSQRT	-
11	10	1100	1	UNALLOCATED	-
	11	11xx		VCVT (between floating-point and integer, Advanced SIMD)	-

Advanced SIMD duplicate (scalar)

These instructions are under [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 opc	Instruction Details
000	VDUP (scalar)
001	UNALLOCATED
01x	UNALLOCATED
1xx	UNALLOCATED

Advanced SIMD three registers of different lengths

These instructions are under [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 U	opc	Instruction Details
	0000	VADDL
	0001	VADDW
	0010	VSUBL
0	0100	VADDHN
	0011	VSUBW
0	0110	VSUBHN
0	1001	VQDMLAL
	0101	VABAL
0	1011	VQDMLSL
0	1101	VQDMULL
	0111	VABDL (integer)
	1000	VMLAL (integer)
	1010	VMSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)
1	1001	UNALLOCATED
1	1011	UNALLOCATED

Decode fields		Instruction Details
U	opc	
1	1101	UNALLOCATED
	1111	UNALLOCATED

Advanced SIMD two registers and a scalar

These instructions are under [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	VMLA (by scalar)	-
0	0011	VQDMLAL	-
	0010	VMLAL (by scalar)	-
0	0111	VQDMLSL	-
	010x	VMLS (by scalar)	-
0	1011	VQDMULL	-
	0110	VMLSL (by scalar)	-
	100x	VMUL (by scalar)	-
1	0011	UNALLOCATED	-
	1010	VMULL (by scalar)	-
1	0111	UNALLOCATED	-
	1100	VQDMULH	-
	1101	VQDMLAH	-
1	1011	UNALLOCATED	-
	1110	VQDMLAH	FEAT_RDM
	1111	VQDMLSH	FEAT_RDM

Advanced SIMD shifts and immediate generation

These instructions are under [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		
000XXXXXXXXXXXX0		Advanced SIMD one register and modified immediate
!= 000XXXXXXXXXXXX0		Advanced SIMD two registers and shift amount

Advanced SIMD one register and modified immediate

These instructions are under [Advanced SIMD shifts and immediate generation.](#)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	i	1	1	1	1	1	D	0	0	0	imm3		Vd		cmode	0	Q	op	1		imm4								

Decode fields		Instruction Details
cmode	op	
0xx0	0	VMOV (immediate) – T1
0xx0	1	VMVN (immediate) – T1
0xx1	0	VORR (immediate) – T1
0xx1	1	VBIC (immediate) – T1
10x0	0	VMOV (immediate) – T3
10x0	1	VMVN (immediate) – T2
10x1	0	VORR (immediate) – T2
10x1	1	VBIC (immediate) – T2
11xx	0	VMOV (immediate) – T4
110x	1	VMVN (immediate) – T3
1110	1	VMOV (immediate) – T5
1111	1	UNALLOCATED

Advanced SIMD two registers and shift amount

These instructions are under [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 != 000xxxxxxxxxxx0

Decode fields					Instruction Details
U	imm3H:L	imm3L	opc	Q	
	!= 0000		0000		VSHR
	!= 0000		0001		VSRA
	!= 0000	000	1010	0	VMOVL
	!= 0000		0010		VRSR
	!= 0000		0011		VRSRA
	!= 0000		0111		VQSHL, VQSHLU (immediate) – VQSHL
	!= 0000		1001	0	VQSHRN, VQSHRUN – VQSHRN
	!= 0000		1001	1	VQRSHRN, VQRSHRUN – VQRSHRN
	!= 0000		1010	0	VSHLL
	!= 0000		11xx		VCVT (between floating-point and fixed-point, Advanced SIMD)
0	!= 0000		0101		VSHL (immediate)
0	!= 0000		1000	0	VSHRN
0	!= 0000		1000	1	VRSHRN
1	!= 0000		0100		VSRI
1	!= 0000		0101		VSLI
1	!= 0000		0110		VQSHL, VQSHLU (immediate) – VQSHLU
1	!= 0000		1000	0	VQSHRN, VQSHRUN – VQSHRUN
1	!= 0000		1000	1	VQRSHRN, VQRSHRUN – VQRSHRUN

Advanced SIMD and System register load/store and 64-bit move

These instructions are under [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	Advanced SIMD and floating-point 64-bit move
00x0	11	System register 64-bit move
!= 00x0	0x	Advanced SIMD and floating-point load/store
!= 00x0	11	System register Load/Store
	10	UNALLOCATED

Advanced SIMD and floating-point 64-bit move

These instructions are under [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	VMOV (between two general-purpose registers and two single-precision registers) – from general-purpose registers
1	0	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) – from general-purpose registers
1			1x		UNALLOCATED
1	1	10	00	1	VMOV (between two general-purpose registers and two single-precision registers) – to general-purpose registers
1	1	11	00	1	VMOV (between two general-purpose registers and a doubleword floating-point register) – to general-purpose registers

System register 64-bit move

These instructions are under [Advanced SIMD and System register load/store and 64-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	0	0	0	D	0	L	Rt2					Rt					1	1	1	cp15	opc1			CRm		

Decode fields		Instruction Details
D	L	
0		UNALLOCATED
1	0	MCRR
1	1	MRRC

Advanced SIMD and floating-point load/store

These instructions are under [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

P	U	W	Decode fields			imm8	Instruction Details	Feature
			L	Rn	size			
0	0	1					UNALLOCATED	-
0	1				0x		UNALLOCATED	-
0	1		0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
0	1		0		11	XXXXXXXX0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
0	1		0		11	XXXXXXXX1	FSTMDBX, FSTMIAX — Increment After	-
0	1		1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
0	1		1		11	XXXXXXXX0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
0	1		1		11	XXXXXXXX1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After	-
1		0	0		01		VSTR — half-precision scalar	FEAT_FP16
1		0	0		10		VSTR — single-precision scalar	-
1		0	0		11		VSTR — double-precision scalar	-
1		0	1	!= 1111	01		VLDR (immediate) — half-precision scalar	FEAT_FP16
1		0	1	!= 1111	10		VLDR (immediate) — single-precision scalar	-
1		0	1	!= 1111	11		VLDR (immediate) — double-precision scalar	-
1	0	1			0x		UNALLOCATED	-
1	0	1	0		10		VSTM, VSTMDB, VSTMIA — single-precision scalar	-
1	0	1	0		11	XXXXXXXX0	VSTM, VSTMDB, VSTMIA — double-precision scalar	-
1	0	1	0		11	XXXXXXXX1	FSTMDBX, FSTMIAX — Decrement Before	-
1	0	1	1		10		VLDM, VLDMDB, VLDMIA — single-precision scalar	-
1	0	1	1		11	XXXXXXXX0	VLDM, VLDMDB, VLDMIA — double-precision scalar	-
1	0	1	1		11	XXXXXXXX1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before	-
1		0	1	1111	01		VLDR (literal) — half-precision scalar	FEAT_FP16
1		0	1	1111	10		VLDR (literal) — single-precision scalar	-
1		0	1	1111	11		VLDR (literal) — double-precision scalar	-
1	1	1					UNALLOCATED	-

System register Load/Store

These instructions are under [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			Decode fields			Instruction Details		
D	L	Rn	CRd	cp15				
!= 000			!= 0101	0	UNALLOCATED			
!= 000	0	1	1111	0101	0	LDC (literal)		

P:U:W	D	L	Decode fields Rn	CRd	cp15	Instruction Details
!= 000					1	UNALLOCATED
!= 000	1			0101	0	UNALLOCATED
0x1	0	0		0101	0	STC – post-indexed
0x1	0	1	!= 1111	0101	0	LDC (immediate) – post-indexed
010	0	0		0101	0	STC – unindexed
010	0	1	!= 1111	0101	0	LDC (immediate) – unindexed
1x0	0	0		0101	0	STC – offset
1x0	0	1	!= 1111	0101	0	LDC (immediate) – offset
1x1	0	0		0101	0	STC – pre-indexed
1x1	0	1	!= 1111	0101	0	LDC (immediate) – pre-indexed

Advanced SIMD and System register 32-bit move

These instructions are under [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 op0 op1		Instruction details	Architecture version
000	000	UNALLOCATED	-
000	001	VMOV (between general-purpose register and half-precision)	FEAT_FP16
000	010	VMOV (between general-purpose register and single-precision)	-
001	010	UNALLOCATED	-
01x	010	UNALLOCATED	-
10x	010	UNALLOCATED	-
110	010	UNALLOCATED	-
111	010	Floating-point move special register	-
	011	Advanced SIMD 8/16/32-bit element move/duplicate	-
	10x	UNALLOCATED	-
	11x	System register 32-bit move	-

Floating-point move special register

These instructions are under [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 L	Instruction Details
0	VMSR
1	VMRS

Advanced SIMD 8/16/32-bit element move/duplicate

These instructions are under [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	
0xx	0		VMOV (general-purpose register to scalar)
	1		VMOV (scalar to general-purpose register)
1xx	0	0x	VDUP (general-purpose register)
1xx	0	1x	UNALLOCATED

System register 32-bit move

These instructions are under [Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	opc1	L	CRn	Rt	1	1	1	cp15	opc2	1	CRm													

Decode fields		Instruction Details
L		
0		MCR
1		MRC

Floating-point data-processing

These instructions are under [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	
1x11	1	Floating-point data-processing (two registers)
1x11	0	Floating-point move immediate
!= 1x11		Floating-point data-processing (three registers)

Floating-point data-processing (two registers)

These instructions are under [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	VABS — half-precision scalar	FEAT_FP16
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	10	1	VABS — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	000	11	1	VABS — double-precision scalar	-
0	001	01	0	VNEG — half-precision scalar	FEAT_FP16
0	001	01	1	VSQRT — half-precision scalar	FEAT_FP16
0	001	10	0	VNEG — single-precision scalar	-
0	001	10	1	VSQRT — single-precision scalar	-
0	001	11	0	VNEG — double-precision scalar	-
0	001	11	1	VSQRT — double-precision scalar	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
0	010	01		UNALLOCATED	-
0	010	10	0	VCVTB — half-precision to single-precision	-
0	010	10	1	VCVTT — half-precision to single-precision	-
0	010	11	0	VCVTB — half-precision to double-precision	-
0	010	11	1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	FEAT_AA32BF16
0	011	01	1	VCVTT (BFloat16)	FEAT_AA32BF16
0	011	10	0	VCVTB — single-precision to half-precision	-
0	011	10	1	VCVTT — single-precision to half-precision	-
0	011	11	0	VCVTB — double-precision to half-precision	-
0	011	11	1	VCVTT — double-precision to half-precision	-
0	100	01	0	VCMP	FEAT_FP16
0	100	01	1	VCMPE	FEAT_FP16
0	100	10	0	VCMP	-
0	100	10	1	VCMPE	-
0	100	11	0	VCMP	-
0	100	11	1	VCMPE	-
0	101	01	0	VCMP	FEAT_FP16
0	101	01	1	VCMPE	FEAT_FP16
0	101	10	0	VCMP	-
0	101	10	1	VCMPE	-
0	101	11	0	VCMP	-
0	101	11	1	VCMPE	-
0	110	01	0	VRINTR — half-precision scalar	FEAT_FP16
0	110	01	1	VRINTZ (floating-point) — half-precision scalar	FEAT_FP16
0	110	10	0	VRINTR — single-precision scalar	-
0	110	10	1	VRINTZ (floating-point) — single-precision scalar	-
0	110	11	0	VRINTR — double-precision scalar	-
0	110	11	1	VRINTZ (floating-point) — double-precision scalar	-
0	111	01	0	VRINTX (floating-point) — half-precision scalar	FEAT_FP16
0	111	01	1	UNALLOCATED	-
0	111	10	0	VRINTX (floating-point) — single-precision scalar	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	0	VRINTX (floating-point) — double-precision scalar	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000	01		VCVT (integer to floating-point, floating-point) — half-precision scalar	FEAT_FP16
1	000	10		VCVT (integer to floating-point, floating-point) — single-precision scalar	-
1	000	11		VCVT (integer to floating-point, floating-point) — double-precision scalar	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	FEAT_JSCVT
1	01x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	01x	10		VCVT (between floating-point and fixed-point, floating-point)	-

Decode fields				Instruction Details	Feature
o1	opc2	size	o3		
1	01x	11		VCVT (between floating-point and fixed-point, floating-point)	-
1	100	01	0	VCVTR	FEAT_FP16
1	100	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	100	10	0	VCVTR	-
1	100	10	1	VCVT (floating-point to integer, floating-point)	-
1	100	11	0	VCVTR	-
1	100	11	1	VCVT (floating-point to integer, floating-point)	-
1	101	01	0	VCVTR	FEAT_FP16
1	101	01	1	VCVT (floating-point to integer, floating-point)	FEAT_FP16
1	101	10	0	VCVTR	-
1	101	10	1	VCVT (floating-point to integer, floating-point)	-
1	101	11	0	VCVTR	-
1	101	11	1	VCVT (floating-point to integer, floating-point)	-
1	11x	01		VCVT (between floating-point and fixed-point, floating-point)	FEAT_FP16
1	11x	10		VCVT (between floating-point and fixed-point, floating-point)	-
1	11x	11		VCVT (between floating-point and fixed-point, floating-point)	-

Floating-point move immediate

These instructions are under [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 size	Instruction Details	Feature
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	FEAT_FP16
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Floating-point data-processing (three registers)

These instructions are under [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	VMLA (floating-point) — half-precision scalar	FEAT_FP16
000	01	1	VMLS (floating-point) — half-precision scalar	FEAT_FP16
000	10	0	VMLA (floating-point) — single-precision scalar	-
000	10	1	VMLS (floating-point) — single-precision scalar	-
000	11	0	VMLA (floating-point) — double-precision scalar	-
000	11	1	VMLS (floating-point) — double-precision scalar	-
001	01	0	VNMLS — half-precision scalar	FEAT_FP16

Decode fields			Instruction Details	Feature
o0:o1	size	o2		
001	01	1	VNMLA — half-precision scalar	FEAT_FP16
001	10	0	VNMLS — single-precision scalar	-
001	10	1	VNMLA — single-precision scalar	-
001	11	0	VNMLS — double-precision scalar	-
001	11	1	VNMLA — double-precision scalar	-
010	01	0	VMUL (floating-point) — half-precision scalar	FEAT_FP16
010	01	1	VNMUL — half-precision scalar	FEAT_FP16
010	10	0	VMUL (floating-point) — single-precision scalar	-
010	10	1	VNMUL — single-precision scalar	-
010	11	0	VMUL (floating-point) — double-precision scalar	-
010	11	1	VNMUL — double-precision scalar	-
011	01	0	VADD (floating-point) — half-precision scalar	FEAT_FP16
011	01	1	VSUB (floating-point) — half-precision scalar	FEAT_FP16
011	10	0	VADD (floating-point) — single-precision scalar	-
011	10	1	VSUB (floating-point) — single-precision scalar	-
011	11	0	VADD (floating-point) — double-precision scalar	-
011	11	1	VSUB (floating-point) — double-precision scalar	-
100	01	0	VDIV — half-precision scalar	FEAT_FP16
100	10	0	VDIV — single-precision scalar	-
100	11	0	VDIV — double-precision scalar	-
101	01	0	VFNMS — half-precision scalar	FEAT_FP16
101	01	1	VFNMA — half-precision scalar	FEAT_FP16
101	10	0	VFNMS — single-precision scalar	-
101	10	1	VFNMA — single-precision scalar	-
101	11	0	VFNMS — double-precision scalar	-
101	11	1	VFNMA — double-precision scalar	-
110	01	0	VFMA — half-precision scalar	FEAT_FP16
110	01	1	VFMS — half-precision scalar	FEAT_FP16
110	10	0	VFMA — single-precision scalar	-
110	10	1	VFMS — single-precision scalar	-
110	11	0	VFMA — double-precision scalar	-
110	11	1	VFMS — double-precision scalar	-

Additional Advanced SIMD and floating-point instructions

These instructions are under [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: $op0<2:1> \neq 11$

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0xx			0x			Advanced SIMD three registers of the same length extension
100		0	!= 00	0	0	Floating-point conditional select

101	00xxxx	0	!= 00		0	Floating-point minNum/maxNum
101	110000	0	!= 00	1	0	Floating-point extraction and insertion
101	111xxx	0	!= 00	1	0	Floating-point directed convert to integer
10x		0	00			Advanced SIMD and floating-point multiply with accumulate
10x		1	0x			Advanced SIMD and floating-point dot product

Advanced SIMD three registers of the same length extension

These instructions are under [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	VCADD — 64-bit SIMD vector		FEAT_FCMA
x1	0x	0	0	0	1	UNALLOCATED		-
x1	0x	0	0	1	0	VCADD — 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	VMMLA		FEAT_AA32BF16
00	00	1	0	1	1	UNALLOCATED		-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector		FEAT_AA32BF16
00	00	1	1	0	1	UNALLOCATED		-
00	00	1	1	1	0	VDOT (vector) — 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	VFMAL (vector)		FEAT_FHM
00	10	0	1			UNALLOCATED		-
00	10	1	0	0		UNALLOCATED		-
00	10	1	0	1	0	VSMMLA		FEAT_AA32I8MM
00	10	1	0	1	1	VUMMLA		FEAT_AA32I8MM
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector		FEAT_DotProd
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector		FEAT_DotProd
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector		FEAT_DotProd
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector		FEAT_DotProd
00	11	0	0		1	VFMA, VFMA, VFMA (BFloat16, vector)		FEAT_AA32BF16
00	11	0	1			UNALLOCATED		-
00	11	1	0			UNALLOCATED		-
00	11	1	1			UNALLOCATED		-
01	10	0	0		1	VFMSL (vector)		FEAT_FHM
01	10	0	1			UNALLOCATED		-
01	10	1	0	0		UNALLOCATED		-

Decode fields						Instruction Details	Feature
op1	op2	op3	op4	Q	U		
01	10	1	0	1	0	VUSMMLA	FEAT_AA32I8MM
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	FEAT_AA32I8MM
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 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	VCMLA	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

These instructions are under [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		Instruction Details	Feature
size	op		
01		VSELEQ, VSELGE, VSELGT, VSELVS — half-precision scalar	FEAT_FP16
10		VSELEQ, VSELGE, VSELGT, VSELVS — single-precision scalar	-
11		VSELEQ, VSELGE, VSELGT, VSELVS — double-precision scalar	-

Floating-point minNum/maxNum

These instructions are under [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		Instruction Details	Feature
size	op		
01	0	VMAXNM — half-precision scalar	FEAT_FP16
01	1	VMINNM — half-precision scalar	FEAT_FP16
10	0	VMAXNM — single-precision scalar	-
10	1	VMINNM — single-precision scalar	-
11	0	VMAXNM — double-precision scalar	-
11	1	VMINNM — double-precision scalar	-

Floating-point extraction and insertion

These instructions are under [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	VMOVX	FEAT_FP16
10	1	VINS	FEAT_FP16
11		UNALLOCATED	-

Floating-point directed convert to integer

These instructions are under [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

o1	Decode fields RM	size	op	Instruction Details	Feature
0		!= 00	1	UNALLOCATED	-
0	00	01	0	VRINTA (floating-point) — half-precision scalar	FEAT_FP16
0	00	10	0	VRINTA (floating-point) — single-precision scalar	-
0	00	11	0	VRINTA (floating-point) — double-precision scalar	-
0	01	01	0	VRINTN (floating-point) — half-precision scalar	FEAT_FP16
0	01	10	0	VRINTN (floating-point) — single-precision scalar	-
0	01	11	0	VRINTN (floating-point) — double-precision scalar	-
0	10	01	0	VRINTP (floating-point) — half-precision scalar	FEAT_FP16
0	10	10	0	VRINTP (floating-point) — single-precision scalar	-
0	10	11	0	VRINTP (floating-point) — double-precision scalar	-
0	11	01	0	VRINTM (floating-point) — half-precision scalar	FEAT_FP16
0	11	10	0	VRINTM (floating-point) — single-precision scalar	-
0	11	11	0	VRINTM (floating-point) — double-precision scalar	-
1	00	01		VCVTA (floating-point) — half-precision scalar	FEAT_FP16
1	00	10		VCVTA (floating-point) — single-precision scalar	-
1	00	11		VCVTA (floating-point) — double-precision scalar	-
1	01	01		VCVTN (floating-point) — half-precision scalar	FEAT_FP16
1	01	10		VCVTN (floating-point) — single-precision scalar	-
1	01	11		VCVTN (floating-point) — double-precision scalar	-
1	10	01		VCVTP (floating-point) — half-precision scalar	FEAT_FP16
1	10	10		VCVTP (floating-point) — single-precision scalar	-
1	10	11		VCVTP (floating-point) — double-precision scalar	-
1	11	01		VCVTM (floating-point) — half-precision scalar	FEAT_FP16
1	11	10		VCVTM (floating-point) — single-precision scalar	-

Decode fields				Instruction Details				Feature
o1	RM	size	op					
1	11	11		VCVTM (floating-point) — double-precision scalar				-

Advanced SIMD and floating-point multiply with accumulate

These instructions are under [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	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point				FEAT_FCMA
0	00		1	VFMAL (by scalar)				FEAT_FHM
0	01		1	VFMSL (by scalar)				FEAT_FHM
0	10		1	UNALLOCATED				-
0	11		1	VFMA, VFMA, VFMA (BFloat16, by scalar)				FEAT_AA32BF16
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point				FEAT_FCMA
1			1	UNALLOCATED				-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point				FEAT_FCMA

Advanced SIMD and floating-point dot product

These instructions are under [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	VDOT (by element) — 64-bit SIMD vector				FEAT_AA32BF16
0	00	1		1	UNALLOCATED				-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector				FEAT_AA32BF16
0	01	0			UNALLOCATED				-
0	10	0			UNALLOCATED				-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector				FEAT_DotProd
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector				FEAT_DotProd
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector				FEAT_DotProd
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector				FEAT_DotProd
0	11				UNALLOCATED				-
1		0			UNALLOCATED				-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector				FEAT_AA32I8MM
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector				FEAT_AA32I8MM
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector				FEAT_AA32I8MM
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector				FEAT_AA32I8MM
1	01	1			UNALLOCATED				-
1	1x	1			UNALLOCATED				-

Load/store multiple

These instructions are under [32-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	0	0	opc	0	W	L	Rn	P	M	register_list																	

Decode fields		Instruction Details
opc	L	
00	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB – T1
00	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB – T1
01	0	STM, STMIA, STMEA
01	1	LDM, LDMIA, LDMFD
10	0	STMDB, STMFD
10	1	LDMDB, LDMEA
11	0	SRS, SRSDA, SRSDB, SRSIA, SRSIB – T2
11	1	RFE, RFEDA, RFEDB, RFEIA, RFEIB – T2

Load/store dual, load/store exclusive, load-acquire/store-release, and table branch

These instructions are under [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				Load/store exclusive
0110	0		000	UNALLOCATED
0110	1		000	TBB, TBH
0110			01x	Load/store exclusive byte/half/dual
0110			1xx	Load-acquire / Store-release
0x11		!= 1111		Load/store dual (immediate, post-indexed)
1x10		!= 1111		Load/store dual (immediate)
1x11		!= 1111		Load/store dual (immediate, pre-indexed)
!= 0xx0		1111		LDRD (literal)

Load/store exclusive

These instructions are under [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		STREX
1		LDREX

Load/store exclusive byte/half/dual

These instructions are under [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		STREXB
0	01		STREXH
0	10		UNALLOCATED
0	11		STREXD
1	00		LDREXB
1	01		LDREXH
1	10		UNALLOCATED
1	11		LDREXD

Load-acquire / Store-release

These instructions are under [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	STLB
0	0	01	STLH
0	0	10	STL
0	0	11	UNALLOCATED
0	1	00	STLEXB
0	1	01	STLEXH
0	1	10	STLEX
0	1	11	STLEXD
1	0	00	LDAB
1	0	01	LDAH
1	0	10	LDA
1	0	11	UNALLOCATED
1	1	00	LDAEXB
1	1	01	LDAEXH
1	1	10	LDAEX
1	1	11	LDAEXD

Load/store dual (immediate, post-indexed)

These instructions are under [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		Rn		Rt		Rt2				imm8									

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
L		
0		STRD (immediate)
1		LDRD (immediate)

Load/store dual (immediate)

These instructions are under [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	STRD (immediate)
1	LDRD (immediate)

Load/store dual (immediate, pre-indexed)

These instructions are under [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
L	
0	STRD (immediate)
1	LDRD (immediate)

Data-processing (shifted register)

These instructions are under [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																

op1	S	Rn	imm3:imm2:stype	Rd	Instruction Details
0000	0		!= 0000011		AND, ANDS (register) — AND, shift or rotate by value
0000	0		0000011		AND, ANDS (register) — AND, rotate right with extend
0000	1		!= 0000011	!= 1111	AND, ANDS (register) — ANDS, shift or rotate by value
0000	1		!= 0000011	1111	TST (register) — shift or rotate by value
0000	1		0000011	!= 1111	AND, ANDS (register) — ANDS, rotate right with extend
0000	1		0000011	1111	TST (register) — rotate right with extend
0001			!= 0000011		BIC, BICS (register) — shift or rotate by value
0001			0000011		BIC, BICS (register) — rotate right with extend
0010	0	!= 1111	!= 0000011		ORR, ORRS (register) — ORR, shift or rotate by value
0010	0	!= 1111	0000011		ORR, ORRS (register) — ORR, rotate right with extend
0010	0	1111	!= 0000011		MOV, MOVS (register) — MOV, shift or rotate by value

op1	S	Decode fields		Rd	Instruction Details
		Rn	imm3:imm2:stype		
0010	0	1111	0000011		MOV, MOVS (register) — MOV, rotate right with extend
0010	1	!= 1111	!= 0000011		ORR, ORRS (register) — ORRS, shift or rotate by value
0010	1	!= 1111	0000011		ORR, ORRS (register) — ORRS, rotate right with extend
0010	1	1111	!= 0000011		MOV, MOVS (register) — MOVS, shift or rotate by value
0010	1	1111	0000011		MOV, MOVS (register) — MOVS, rotate right with extend
0011	0	!= 1111	!= 0000011		ORN, ORNS (register) — ORN, shift or rotate by value
0011	0	!= 1111	0000011		ORN, ORNS (register) — ORN, rotate right with extend
0011	0	1111	!= 0000011		MVN, MVNS (register) — MVN, shift or rotate by value
0011	0	1111	0000011		MVN, MVNS (register) — MVN, rotate right with extend
0011	1	!= 1111	!= 0000011		ORN, ORNS (register) — ORNS, shift or rotate by value
0011	1	!= 1111	0000011		ORN, ORNS (register) — ORNS, rotate right with extend
0011	1	1111	!= 0000011		MVN, MVNS (register) — MVNS, shift or rotate by value
0011	1	1111	0000011		MVN, MVNS (register) — MVNS, rotate right with extend
0100	0		!= 0000011		EOR, EORS (register) — EOR, shift or rotate by value
0100	0		0000011		EOR, EORS (register) — EOR, rotate right with extend
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value
0100	1		!= 0000011	1111	TEQ (register) — shift or rotate by value
0100	1		0000011	!= 1111	EOR, EORS (register) — EORS, rotate right with extend
0100	1		0000011	1111	TEQ (register) — rotate right with extend
0101					UNALLOCATED
0110	0		xxxxx00		PKHBT, PKHTB — PKHBT
0110	0		xxxxx01		UNALLOCATED
0110	0		xxxxx10		PKHBT, PKHTB — PKHTB
0110	0		xxxxx11		UNALLOCATED
0111					UNALLOCATED
1000	0	!= 1101	!= 0000011		ADD, ADDS (register) — ADD, shift or rotate by value
1000	0	!= 1101	0000011		ADD, ADDS (register) — ADD, rotate right with extend
1000	0	1101	!= 0000011		ADD, ADDS (SP plus register) — ADD, shift or rotate by value
1000	0	1101	0000011		ADD, ADDS (SP plus register) — ADD, rotate right with extend
1000	1		!= 0000011	1111	CMN (register) — shift or rotate by value

op1	S	Decode fields			Rd	Instruction Details
		Rn	imm3:imm2:stype			
1000	1	!= 1101	!= 0000011		!= 1111	ADD, ADDS (register) — ADDS, shift or rotate by value
1000	1	!= 1101	0000011		!= 1111	ADD, ADDS (register) — ADDS, rotate right with extend
1000	1		0000011		1111	CMN (register) — rotate right with extend
1000	1	1101	!= 0000011		!= 1111	ADD, ADDS (SP plus register) — ADDS, shift or rotate by value
1000	1	1101	0000011		!= 1111	ADD, ADDS (SP plus register) — ADDS, rotate right with extend
1001						UNALLOCATED
1010			!= 0000011			ADC, ADCS (register) — shift or rotate by value
1010			0000011			ADC, ADCS (register) — rotate right with extend
1011			!= 0000011			SBC, SBCS (register) — shift or rotate by value
1011			0000011			SBC, SBCS (register) — rotate right with extend
1100						UNALLOCATED
1101	0	!= 1101	!= 0000011			SUB, SUBS (register) — SUB, shift or rotate by value
1101	0	!= 1101	0000011			SUB, SUBS (register) — SUB, rotate right with extend
1101	0	1101	!= 0000011			SUB, SUBS (SP minus register) — SUB, shift or rotate by value
1101	0	1101	0000011			SUB, SUBS (SP minus register) — SUB, rotate right with extend
1101	1		!= 0000011		1111	CMP (register) — shift or rotate by value
1101	1	!= 1101	!= 0000011		!= 1111	SUB, SUBS (register) — SUBS, shift or rotate by value
1101	1	!= 1101	0000011		!= 1111	SUB, SUBS (register) — SUBS, rotate right with extend
1101	1		0000011		1111	CMP (register) — rotate right with extend
1101	1	1101	!= 0000011		!= 1111	SUB, SUBS (SP minus register) — SUBS, shift or rotate by value
1101	1	1101	0000011		!= 1111	SUB, SUBS (SP minus register) — SUBS, rotate right with extend
1110			!= 0000011			RSB, RSBS (register) — shift or rotate by value
1110			0000011			RSB, RSBS (register) — rotate right with extend
1111						UNALLOCATED

Branches and miscellaneous control

These instructions are under [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									

op0	op1	Decode fields				op5	Instruction details
		op2	op3	op4			
0	1110	0x	0x0		0	MSR (register)	
0	1110	0x	0x0		1	MSR (Banked register)	
0	1110	10	0x0	000		Hints	
0	1110	10	0x0	!= 000		Change processor state	

0	1110	11	0x0			Miscellaneous system
0	1111	00	0x0			BXJ
0	1111	01	0x0			Exception return
0	1111	1x	0x0		0	MRS
0	1111	1x	0x0		1	MRS (Banked register)
1	1110	00	000			DCPS
1	1110	00	010			UNALLOCATED
1	1110	01	0x0			UNALLOCATED
1	1110	1x	0x0			UNALLOCATED
1	1111	0x	0x0			UNALLOCATED
1	1111	1x	0x0			Exception generation
	!= 111x		0x0			B – T3
			0x1			B – T4
			1x0			BL, BLX (immediate) – T2
			1x1			BL, BLX (immediate) – T1

Hints

These instructions are under [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	NOP	-
0000	0001	YIELD	-
0000	0010	WFE	-
0000	0011	WFI	-
0000	0100	SEV	-
0000	0101	SEVL	-
0000	011x	Reserved hint, behaves as NOP	-
0000	1xxx	Reserved hint, behaves as NOP	-
0001	0000	ESB	FEAT_RAS
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	TSB CSYNC	FEAT_TRF
0001	0011	Reserved hint, behaves as NOP	-
0001	0100	CSDB	-
0001	0101	Reserved hint, behaves as NOP	-
0001	011x	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		DBG	-

Change processor state

These instructions are under [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	CPS, CPSID, CPSIE – change mode
01		UNALLOCATED
10		CPS, CPSID, CPSIE – interrupt enable and change mode
11		CPS, CPSID, CPSIE – interrupt disable and change mode

Miscellaneous system

These instructions are under [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		CLREX	-
0011		UNALLOCATED	-
0100	!= 0x00	DSB	-
0100	0000	SSBB	-
0100	0100	PSSBB	-
0101		DMB	-
0110		ISB	-
0111		SB	FEAT_SB
1xxx		UNALLOCATED	-

Exception return

These instructions are under [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		SUB, SUBS (immediate)
111000000000		ERET

DCPS

These instructions are under [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		Instruction Details
imm4	imm10	
!= 1111		UNALLOCATED
1111	!= 0000000000	UNALLOCATED

Decode fields				Instruction Details
op1	S	Rn	Rd	
1001				UNALLOCATED
1010				ADC, ADCS (immediate)
1011				SBC, SBCS (immediate)
1100				UNALLOCATED
1101	0	!= 1101		SUB, SUBS (immediate) – SUB
1101	0	1101		SUB, SUBS (SP minus immediate) – SUB
1101	1	!= 1101	!= 1111	SUB, SUBS (immediate) – SUBS
1101	1	1101	!= 1111	SUB, SUBS (SP minus immediate) – SUBS
1101	1		1111	CMP (immediate)
1110				RSB, RSBS (immediate)
1111				UNALLOCATED

Data-processing (plain binary immediate)

These instructions are under [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					1	op0		op1	0							0															

Decode fields		Instruction details
op0	op1	
0	0x	Data-processing (simple immediate)
0	10	Move Wide (16-bit immediate)
0	11	UNALLOCATED
1		Saturate, Bitfield

Data-processing (simple immediate)

These instructions are under [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	ADD, ADDS (immediate)
0	0	1101	ADD, ADDS (SP plus immediate)
0	0	1111	ADR – T3
0	1		UNALLOCATED
1	0		UNALLOCATED
1	1	!= 11x1	SUB, SUBS (immediate)
1	1	1101	SUB, SUBS (SP minus immediate)
1	1	1111	ADR – T2

Move Wide (16-bit immediate)

These instructions are under [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												

L	Decode fields		Instruction Details
	itype	Rm	
0	0011	!= 11x1	VST2 (multiple 2-element structures)
0	0011	1101	VST2 (multiple 2-element structures)
0	0011	1111	VST2 (multiple 2-element structures)
0	010x	!= 11x1	VST3 (multiple 3-element structures)
0	010x	1101	VST3 (multiple 3-element structures)
0	010x	1111	VST3 (multiple 3-element structures)
0	0110	!= 11x1	VST1 (multiple single elements)
0	0110	1101	VST1 (multiple single elements)
0	0110	1111	VST1 (multiple single elements)
0	0111	!= 11x1	VST1 (multiple single elements)
0	0111	1101	VST1 (multiple single elements)
0	0111	1111	VST1 (multiple single elements)
0	100x	!= 11x1	VST2 (multiple 2-element structures)
0	100x	1101	VST2 (multiple 2-element structures)
0	100x	1111	VST2 (multiple 2-element structures)
0	1010	!= 11x1	VST1 (multiple single elements)
0	1010	1101	VST1 (multiple single elements)
0	1010	1111	VST1 (multiple single elements)
1	000x	!= 11x1	VLD4 (multiple 4-element structures)
1	000x	1101	VLD4 (multiple 4-element structures)
1	000x	1111	VLD4 (multiple 4-element structures)
1	0010	!= 11x1	VLD1 (multiple single elements)
1	0010	1101	VLD1 (multiple single elements)
1	0010	1111	VLD1 (multiple single elements)
1	0011	!= 11x1	VLD2 (multiple 2-element structures)
1	0011	1101	VLD2 (multiple 2-element structures)
1	0011	1111	VLD2 (multiple 2-element structures)
1	010x	!= 11x1	VLD3 (multiple 3-element structures)
1	010x	1101	VLD3 (multiple 3-element structures)
1	010x	1111	VLD3 (multiple 3-element structures)
	1011		UNALLOCATED
1	0110	!= 11x1	VLD1 (multiple single elements)
1	0110	1101	VLD1 (multiple single elements)
1	0110	1111	VLD1 (multiple single elements)
1	0111	!= 11x1	VLD1 (multiple single elements)
1	0111	1101	VLD1 (multiple single elements)
1	0111	1111	VLD1 (multiple single elements)
	11xx		UNALLOCATED
1	100x	!= 11x1	VLD2 (multiple 2-element structures)
1	100x	1101	VLD2 (multiple 2-element structures)
1	100x	1111	VLD2 (multiple 2-element structures)
1	1010	!= 11x1	VLD1 (multiple single elements)
1	1010	1101	VLD1 (multiple single elements)
1	1010	1111	VLD1 (multiple single elements)

Advanced SIMD load single structure to all lanes

These instructions are under [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	VLD1 (single element to all lanes)
1	00		1101	VLD1 (single element to all lanes)
1	00		1111	VLD1 (single element to all lanes)
1	01		!= 11x1	VLD2 (single 2-element structure to all lanes)
1	01		1101	VLD2 (single 2-element structure to all lanes)
1	01		1111	VLD2 (single 2-element structure to all lanes)
1	10	0	!= 11x1	VLD3 (single 3-element structure to all lanes)
1	10	0	1101	VLD3 (single 3-element structure to all lanes)
1	10	0	1111	VLD3 (single 3-element structure to all lanes)
1	10	1		UNALLOCATED
1	11		!= 11x1	VLD4 (single 4-element structure to all lanes)
1	11		1101	VLD4 (single 4-element structure to all lanes)
1	11		1111	VLD4 (single 4-element structure to all lanes)

Advanced SIMD load/store single structure to one lane

These instructions are under [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	VST1 (single element from one lane)
0	00	00	1101	VST1 (single element from one lane)
0	00	00	1111	VST1 (single element from one lane)
0	00	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	00	01	1101	VST2 (single 2-element structure from one lane)
0	00	01	1111	VST2 (single 2-element structure from one lane)
0	00	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	00	10	1101	VST3 (single 3-element structure from one lane)
0	00	10	1111	VST3 (single 3-element structure from one lane)
0	00	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	00	11	1101	VST4 (single 4-element structure from one lane)
0	00	11	1111	VST4 (single 4-element structure from one lane)
0	01	00	!= 11x1	VST1 (single element from one lane)
0	01	00	1101	VST1 (single element from one lane)
0	01	00	1111	VST1 (single element from one lane)
0	01	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	01	01	1101	VST2 (single 2-element structure from one lane)
0	01	01	1111	VST2 (single 2-element structure from one lane)
0	01	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	01	10	1101	VST3 (single 3-element structure from one lane)

L	Decode fields			Instruction Details
	size	N	Rm	
0	01	10	1111	VST3 (single 3-element structure from one lane)
0	01	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	01	11	1101	VST4 (single 4-element structure from one lane)
0	01	11	1111	VST4 (single 4-element structure from one lane)
0	10	00	!= 11x1	VST1 (single element from one lane)
0	10	00	1101	VST1 (single element from one lane)
0	10	00	1111	VST1 (single element from one lane)
0	10	01	!= 11x1	VST2 (single 2-element structure from one lane)
0	10	01	1101	VST2 (single 2-element structure from one lane)
0	10	01	1111	VST2 (single 2-element structure from one lane)
0	10	10	!= 11x1	VST3 (single 3-element structure from one lane)
0	10	10	1101	VST3 (single 3-element structure from one lane)
0	10	10	1111	VST3 (single 3-element structure from one lane)
0	10	11	!= 11x1	VST4 (single 4-element structure from one lane)
0	10	11	1101	VST4 (single 4-element structure from one lane)
0	10	11	1111	VST4 (single 4-element structure from one lane)
1	00	00	!= 11x1	VLD1 (single element to one lane)
1	00	00	1101	VLD1 (single element to one lane)
1	00	00	1111	VLD1 (single element to one lane)
1	00	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	00	01	1101	VLD2 (single 2-element structure to one lane)
1	00	01	1111	VLD2 (single 2-element structure to one lane)
1	00	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	00	10	1101	VLD3 (single 3-element structure to one lane)
1	00	10	1111	VLD3 (single 3-element structure to one lane)
1	00	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	00	11	1101	VLD4 (single 4-element structure to one lane)
1	00	11	1111	VLD4 (single 4-element structure to one lane)
1	01	00	!= 11x1	VLD1 (single element to one lane)
1	01	00	1101	VLD1 (single element to one lane)
1	01	00	1111	VLD1 (single element to one lane)
1	01	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	01	01	1101	VLD2 (single 2-element structure to one lane)
1	01	01	1111	VLD2 (single 2-element structure to one lane)
1	01	10	!= 11x1	VLD3 (single 3-element structure to one lane)
1	01	10	1101	VLD3 (single 3-element structure to one lane)
1	01	10	1111	VLD3 (single 3-element structure to one lane)
1	01	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	01	11	1101	VLD4 (single 4-element structure to one lane)
1	01	11	1111	VLD4 (single 4-element structure to one lane)
1	10	00	!= 11x1	VLD1 (single element to one lane)
1	10	00	1101	VLD1 (single element to one lane)
1	10	00	1111	VLD1 (single element to one lane)
1	10	01	!= 11x1	VLD2 (single 2-element structure to one lane)
1	10	01	1101	VLD2 (single 2-element structure to one lane)
1	10	01	1111	VLD2 (single 2-element structure to one lane)
1	10	10	!= 11x1	VLD3 (single 3-element structure to one lane)

Decode fields				Instruction Details
L	size	N	Rm	
1	10	10	1101	VLD3 (single 3-element structure to one lane)
1	10	10	1111	VLD3 (single 3-element structure to one lane)
1	10	11	!= 11x1	VLD4 (single 4-element structure to one lane)
1	10	11	1101	VLD4 (single 4-element structure to one lane)
1	10	11	1111	VLD4 (single 4-element structure to one lane)

Load/store single

These instructions are under [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	Load/store, unsigned (register offset)
00		!= 1111	000001	UNALLOCATED
00		!= 1111	00001x	UNALLOCATED
00		!= 1111	0001xx	UNALLOCATED
00		!= 1111	001xxx	UNALLOCATED
00		!= 1111	01xxxx	UNALLOCATED
00		!= 1111	10x0xx	UNALLOCATED
00		!= 1111	10x1xx	Load/store, unsigned (immediate, post-indexed)
00		!= 1111	1100xx	Load/store, unsigned (negative immediate)
00		!= 1111	1110xx	Load/store, unsigned (unprivileged)
00		!= 1111	11x1xx	Load/store, unsigned (immediate, pre-indexed)
01		!= 1111		Load/store, unsigned (positive immediate)
0x		1111		Load, unsigned (literal)
10	1	!= 1111	000000	Load/store, signed (register offset)
10	1	!= 1111	000001	UNALLOCATED
10	1	!= 1111	00001x	UNALLOCATED
10	1	!= 1111	0001xx	UNALLOCATED
10	1	!= 1111	001xxx	UNALLOCATED
10	1	!= 1111	01xxxx	UNALLOCATED
10	1	!= 1111	10x0xx	UNALLOCATED
10	1	!= 1111	10x1xx	Load/store, signed (immediate, post-indexed)
10	1	!= 1111	1100xx	Load/store, signed (negative immediate)
10	1	!= 1111	1110xx	Load/store, signed (unprivileged)
10	1	!= 1111	11x1xx	Load/store, signed (immediate, pre-indexed)
11	1	!= 1111		Load/store, signed (positive immediate)
1x	1	1111		Load, signed (literal)

Load/store, unsigned (register offset)

These instructions are under [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		STRB (register)
00	1	!= 1111	LDRB (register)
00	1	1111	PLD, PLDW (register) — preload read
01	0		STRH (register)
01	1	!= 1111	LDRH (register)
01	1	1111	PLD, PLDW (register) — preload write
10	0		STR (register)
10	1		LDR (register)
11			UNALLOCATED

Load/store, unsigned (immediate, post-indexed)

These instructions are under [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	
00	0	STRB (immediate)
00	1	LDRB (immediate)
01	0	STRH (immediate)
01	1	LDRH (immediate)
10	0	STR (immediate)
10	1	LDR (immediate)
11		UNALLOCATED

Load/store, unsigned (negative immediate)

These instructions are under [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		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (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	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			Instruction Details
size	L	Rt	
00	0		STRB (immediate)
00	1	!= 1111	LDRB (immediate)
00	1	1111	PLD, PLDW (immediate) — preload read
01	0		STRH (immediate)
01	1	!= 1111	LDRH (immediate)
01	1	1111	PLD, PLDW (immediate) — preload write
10	0		STR (immediate)
10	1		LDR (immediate)

Load, unsigned (literal)

These instructions are under [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	PLD (literal)
00	1	!= 1111	LDRB (literal)
01	1	!= 1111	LDRH (literal)
10	1		LDR (literal)
11			UNALLOCATED

Load/store, signed (register offset)

These instructions are under [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	Rt	
00	!= 1111	LDRSB (register)
00	1111	PLI (register)
01	!= 1111	LDRSH (register)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Load/store, signed (immediate, post-indexed)

These instructions are under [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 size	Instruction Details
00	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (negative immediate)

These instructions are under [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	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Load/store, signed (unprivileged)

These instructions are under [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	LDRSBT
01	LDRSHT
1x	UNALLOCATED

Load/store, signed (immediate, pre-indexed)

These instructions are under [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	LDRSB (immediate)
01	LDRSH (immediate)
1x	UNALLOCATED

Load/store, signed (positive immediate)

These instructions are under [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 size	Rt	Instruction Details
00	!= 1111	LDRSB (immediate)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (immediate)
01	1111	Reserved hint, behaves as NOP

Load, signed (literal)

These instructions are under [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 size	Rt	Instruction Details
00	!= 1111	LDRSB (literal)
00	1111	PLI (immediate, literal)
01	!= 1111	LDRSH (literal)
01	1111	Reserved hint, behaves as NOP
1x		UNALLOCATED

Data-processing (register)

These instructions are under [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 op0	op1	op2	Instruction details
0	1111	0000	MOV, MOVS (register-shifted register) – T2, Flag setting
0	1111	0001	UNALLOCATED
0	1111	001x	UNALLOCATED
0	1111	01xx	UNALLOCATED
0	1111	1xxx	Register extends
1	1111	0xxx	Parallel add-subtract
1	1111	10xx	Data-processing (two source registers)

1	1111	11xx	UNALLOCATED
	!= 1111		UNALLOCATED

Register extends

These instructions are under [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	SXTAH
00	0	1111	SXTH
00	1	!= 1111	UXTAH
00	1	1111	UXTH
01	0	!= 1111	SXTAB16
01	0	1111	SXTB16
01	1	!= 1111	UXTAB16
01	1	1111	UXTB16
10	0	!= 1111	SXTAB
10	0	1111	SXTB
10	1	!= 1111	UXTAB
10	1	1111	UXTB
11			UNALLOCATED

Parallel add-subtract

These instructions are under [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	SADD8
000	0	0	1	QADD8
000	0	1	0	SHADD8
000	0	1	1	UNALLOCATED
000	1	0	0	UADD8
000	1	0	1	UQADD8
000	1	1	0	UHADD8
000	1	1	1	UNALLOCATED
001	0	0	0	SADD16
001	0	0	1	QADD16
001	0	1	0	SHADD16
001	0	1	1	UNALLOCATED
001	1	0	0	UADD16
001	1	0	1	UQADD16
001	1	1	0	UHADD16
001	1	1	1	UNALLOCATED
010	0	0	0	SASX

Decode fields				Instruction Details
op1	U	H	S	
010	0	0	1	QASX
010	0	1	0	SHASX
010	0	1	1	UNALLOCATED
010	1	0	0	UASX
010	1	0	1	UQASX
010	1	1	0	UHASX
010	1	1	1	UNALLOCATED
100	0	0	0	SSUB8
100	0	0	1	QSUB8
100	0	1	0	SHSUB8
100	0	1	1	UNALLOCATED
100	1	0	0	USUB8
100	1	0	1	UQSUB8
100	1	1	0	UHSUB8
100	1	1	1	UNALLOCATED
101	0	0	0	SSUB16
101	0	0	1	QSUB16
101	0	1	0	SHSUB16
101	0	1	1	UNALLOCATED
101	1	0	0	USUB16
101	1	0	1	UQSUB16
101	1	1	0	UHSUB16
101	1	1	1	UNALLOCATED
110	0	0	0	SSAX
110	0	0	1	QSAX
110	0	1	0	SHSAX
110	0	1	1	UNALLOCATED
110	1	0	0	USAX
110	1	0	1	UQSAX
110	1	1	0	UHSAX
110	1	1	1	UNALLOCATED
111				UNALLOCATED

Data-processing (two source registers)

These instructions are under [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
op1	op2	
000	00	QADD
000	01	QDADD
000	10	QSUB
000	11	QDSUB
001	00	REV
001	01	REV16
001	10	RBIT

Decode fields		Instruction Details
op1	op2	
001	11	REVSH
010	00	SEL
010	01	UNALLOCATED
010	1x	UNALLOCATED
011	00	CLZ
011	01	UNALLOCATED
011	1x	UNALLOCATED
100	00	CRC32 – CRC32B
100	01	CRC32 – CRC32H
100	10	CRC32 – CRC32W
100	11	CONSTRAINED UNPREDICTABLE
101	00	CRC32C – CRC32CB
101	01	CRC32C – CRC32CH
101	10	CRC32C – CRC32CW
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

These instructions are under [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		Multiply and absolute difference
01		UNALLOCATED
1x		UNALLOCATED

Multiply and absolute difference

These instructions are under [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			Instruction Details
op1	Ra	op2	
000	!= 1111	00	MLA, MLAS
000		01	MLS
000		1x	UNALLOCATED
000	1111	00	MUL, MULS
001	!= 1111	00	SMLABB, SMLABT, SMLATB, SMLATT – SMLABB
001	!= 1111	01	SMLABB, SMLABT, SMLATB, SMLATT – SMLABT
001	!= 1111	10	SMLABB, SMLABT, SMLATB, SMLATT – SMLATB
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT – SMLATT

Decode fields			Instruction Details
op1	Ra	op2	
001	1111	00	SMULBB, SMULBT, SMULTB, SMULTT – SMULBB
001	1111	01	SMULBB, SMULBT, SMULTB, SMULTT – SMULBT
001	1111	10	SMULBB, SMULBT, SMULTB, SMULTT – SMULTB
001	1111	11	SMULBB, SMULBT, SMULTB, SMULTT – SMULTT
010	!= 1111	00	SMLAD, SMLADX – SMLAD
010	!= 1111	01	SMLAD, SMLADX – SMLADX
010		1x	UNALLOCATED
010	1111	00	SMUAD, SMUADX – SMUAD
010	1111	01	SMUAD, SMUADX – SMUADX
011	!= 1111	00	SMLAWB, SMLAWT – SMLAWB
011	!= 1111	01	SMLAWB, SMLAWT – SMLAWT
011		1x	UNALLOCATED
011	1111	00	SMULWB, SMULWT – SMULWB
011	1111	01	SMULWB, SMULWT – SMULWT
100	!= 1111	00	SMLSD, SMLSX – SMLSD
100	!= 1111	01	SMLSD, SMLSX – SMLSX
100		1x	UNALLOCATED
100	1111	00	SMUSD, SMUSDX – SMUSD
100	1111	01	SMUSD, SMUSDX – SMUSDX
101	!= 1111	00	SMMLA, SMMLAR – SMMLA
101	!= 1111	01	SMMLA, SMMLAR – SMMLAR
101		1x	UNALLOCATED
101	1111	00	SMMUL, SMMULR – SMMUL
101	1111	01	SMMUL, SMMULR – SMMULR
110		00	SMMLS, SMMLSR – SMMLS
110		01	SMMLS, SMMLSR – SMMLSR
110		1x	UNALLOCATED
111	!= 1111	00	USADA8
111		01	UNALLOCATED
111		1x	UNALLOCATED
111	1111	00	USAD8

Long multiply and divide

These instructions are under [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		SMULL, SMULLS
001	!= 1111		UNALLOCATED
001	1111		SDIV
010	!= 0000		UNALLOCATED
010	0000		UMULL, UMULLS
011	!= 1111		UNALLOCATED
011	1111		UDIV

Decode fields		Instruction Details
op1	op2	
100	0000	SMLAL, SMLALS
100	0001	UNALLOCATED
100	001x	UNALLOCATED
100	01xx	UNALLOCATED
100	1000	SMLALBB, SMLALBT, SMLALTB, SMLALTT – SMLALBB
100	1001	SMLALBB, SMLALBT, SMLALTB, SMLALTT – SMLALBT
100	1010	SMLALBB, SMLALBT, SMLALTB, SMLALTT – SMLALTB
100	1011	SMLALBB, SMLALBT, SMLALTB, SMLALTT – SMLALTT
100	1100	SMLALD, SMLALDX – SMLALD
100	1101	SMLALD, SMLALDX – SMLALDX
100	111x	UNALLOCATED
101	0xxx	UNALLOCATED
101	10xx	UNALLOCATED
101	1100	SMLS LD, SMLS LD X – SMLS LD
101	1101	SMLS LD, SMLS LD X – SMLS LD X
101	111x	UNALLOCATED
110	0000	UMLAL, UMLALS
110	0001	UNALLOCATED
110	001x	UNALLOCATED
110	010x	UNALLOCATED
110	0110	UMAAL
110	0111	UNALLOCATED
110	1xxx	UNALLOCATED
111		UNALLOCATED

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46

Copyright © 2010-2022 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


```

// 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
    elsif stage == TranslationStage_1 || (stage == TranslationStage_12 && !HaveEL(EL2)) then
        stage = TranslationStage_1;
        ss = SecurityStateAtEL(PSTATE.EL);
        regime = if ss == SS_Secure && ELUsingAArch32(EL3) then Regime_EL30 else Regime_EL10;
        eae = TTBCR.EAE == '1';

    // ATS12NS0xx instructions
    else
        regime = Regime_EL10;
        eae = if HaveAArch32EL(EL3) then TTBCR_NS.EAE == '1' else TTBCR.EAE == '1';
        ss = SS_NonSecure;

    AddressDescriptor addrdesc;
    SDFType sdfdtype;
    aligned = TRUE;
    ispriv = el != EL0;
    supersection = '0';
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;

    // Prepare fault fields in case a fault is detected
    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    if eae then
        (fault, addrdesc) = AArch32.S1TranslatelD(fault, regime, ss, vaddress, acctype, aligned,
            iswrite, ispriv);
    else
        (fault, addrdesc, sdfdtype) = AArch32.S1TranslateSD(fault, regime, ss, vaddress, acctype,
            aligned, iswrite, ispriv);
        supersection = if sdfdtype == SDFType_Supersection then '1' else '0';

    // ATS12NS0xx instructions
    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        s2fslwalk = FALSE;
        (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fslwalk, acctype, aligned,
            iswrite, ispriv);

    if fault.statuscode != Fault_None then
        // Take exception when External abort occurs on translation table walk
        if (IsExternalAbort(fault) || (stage == TranslationStage_1 && el != EL2 && PSTATE.EL == EL1
            && EL2Enabled() && fault.s2fslwalk)) then
            PAR = bits(64) UNKNOWN;
            AArch32.Abort(vaddress, 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(PAR.EncodeShareability(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.address.paspace == PAS_Secure then
      ns = '0';
    else
      ns = '1';
    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.address.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(32) 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 = AArch32.FaultStatusLD(TRUE, fault);
  return syndrome<5:0>;
```

Library pseudocode for aarch32/at/AArch32.PARFaultStatusSD

```
// AArch32.PARFaultStatusSD()
// =====
// Fault status field decoding of 32-bit PAR.

bits(6) AArch32.PARFaultStatusSD(FaultRecord fault)
  bits(32) syndrome;

  syndrome = AArch32.FaultStatusSD(TRUE, fault);
  return syndrome<12,10,3:0>;
```

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
        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
        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;
```



```

// AArch32.DC()
// =====
// Perform Data Cache Operation.

AArch32.DC(bits(32) regval, CacheOp cacheop, CacheOpScope opscope)
    AccType acctype = AccType\_DC;
    CacheRecord cache;

    cache.acctype = acctype;
    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.set, cache.way, cache.level) = DecodeSW(ZeroExtend(regval), CacheType\_Data);

        if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
            ((!ELUsingAArch32(EL2) && HCR_EL2.SWIO == '1') || (ELUsingAArch32(EL2) && HCR.SWIO == '1') ||
            (!ELUsingAArch32(EL2) && HCR_EL2.<DC,VM> != '00') || (ELUsingAArch32(EL2) && HCR.<DC,VM> != '00'))
            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);
    iswrite = cacheop == CacheOp\_Invalidate;
    vaddress = regval;

    size = 0; // by default no watchpoint address
    if iswrite then
        size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
        assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
        assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
        vaddress = Align(regval, size);

    cache.translated = need_translate;
    cache.vaddress = ZeroExtend(vaddress);

    if need_translate then
        wasaligned = TRUE;
        memaddrdesc = AArch32.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        memattrs = memaddrdesc.memattrs;
        cache.paddress = memaddrdesc.paddress;
        if opscope == CacheOpScope\_PoC then
            cache.shareability = 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'))
    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
        elsif !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 == HIGH && SPIDEN == HIGH;

```

Library pseudocode for aarch32/debug/breakpoint/AArch32.BreakpointMatch

```
// AArch32.BreakpointMatch()
// =====
// Breakpoint matching in an AArch32 translation regime.

(boolean,boolean) AArch32.BreakpointMatch(integer n, bits(32) vaddress,
                                           integer size)
  assert ELUsingAArch32\(S1TranslationRegime\(\)\);
  assert n < NumBreakpointsImplemented\(\);

  enabled = DBGBCR[n].E == '1';
  ispriv = PSTATE.EL != EL0;
  linked = DBGBCR[n].BT IN {'0x01'};
  isbreakpt = TRUE;
  linked_to = FALSE;

  state_match = AArch32.StateMatch(DBGBCR[n].SSC, DBGBCR[n].HMC, DBGBCR[n].PMC,
                                   linked, DBGBCR[n].LBN, isbreakpt, ispriv);
  (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\(Unpredictable\_BPMISMATCHHALF\);

  match = value_match && state_match && enabled;
  mismatch = value_mismatch && state_match && enabled;

  return (match, mismatch);
```



```

// 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;

// 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
    Constraint c;
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMP);
    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);

context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR[n].BT;
Constraint c;

if ((dbgtype IN {'011x','11xx'} && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
    (dbgtype IN {'010x'} && HaltOnBreakpointOrWatchpoint()) || // Address mismatch
    (!(dbgtype IN {'0x0x'}) && !context_aware) || // Context match
    (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then // EL2 extension
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return (FALSE,FALSE);
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype IN {'0x0x'});
mismatch = (dbgtype IN {'010x'});
match_vmid = (dbgtype IN {'10xx'});
match_cid1 = (dbgtype IN {'xx1x'});
match_cid2 = (dbgtype IN {'11xx'});
linked = (dbgtype IN {'xxx1'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return (FALSE,FALSE);

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return (FALSE,FALSE);

// Do the comparison.
boolean BVR_match;
if match_addr then
    boolean byte_select_match;
    byte = UInt(vaddress<1:0>);
    assert byte IN {0,2}; // "vaddress" is halfword aligned
    byte_select_match = (DBGBCR[n].BAS<byte> == '1');
    integer top = 31;
    BVR_match = (vaddress<top:2> == DBGBCR[n]<top:2>) && byte_select_match;

elseif match_cid1 then
    BVR_match = (PSTATE.EL != EL2 && CONTEXTIDR == DBGBCR[n]<31:0>);
boolean BXVR_match;

```

```

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 !Have16bitVMID\(\) || 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 && (HaveVirtHostExt\(\) || HaveV82Debug\(\)) &&
        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);

```



```

// 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, bits(4) LBN,
                           boolean isbreakpt, boolean ispriv)

// "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.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpt);
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 = isbreakpt && HMC == '0' && PxC == '00' && SSC != '11';

boolean priv_match;
if !ispriv && !isbreakpt then
    priv_match = PL0_match;
elseif SSU_match then
    priv_match = PSTATE.M IN {M32_User,M32_Svc,M32_System};
else
    case PSTATE.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;

boolean security_state_match;
ss = CurrentSecurityState();
case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = ss == SS_NonSecure; // Non-secure only
    when '10' security_state_match = ss == SS_Secure; // Secure only
    when '11' security_state_match = (HMC == '1' || ss == SS_Secure); // HMC=1 -> Both, 0 -> Secure

integer lbn;
if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable_BPNOTCTX);
        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

boolean linked_match;
if linked then
    vaddress = bits(32) UNKNOWN;
    linked_to = TRUE;
    (linked_match,-) = AArch32.BreakpointValueMatch(lbn, vaddress, linked_to);

```

```
return priv_match && security_state_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 == EL0 then enabled = enabled || SDER.SUIDEN == '1';  
        else  
            enabled = from_el != EL2;  
  
    return enabled;
```

Library pseudocode for aarch32/debug/pmu/AArch32.CheckForPMUOverflow

```
// AArch32.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch32.CheckForPMUOverflow()
  if !ELUsingAArch32(EL1) then
    AArch64.CheckForPMUOverflow();
    return;
  bit hpme;
  if HaveEL(EL2) then
    hpme = if !ELUsingAArch32(EL2) then MDCR_EL2.HPME else HDCR.HPME;
  boolean pmuirq;
  bit E;
  pmuirq = PMCR.E == '1' && PMINTENSET.C == '1' && PMOVSSET.C == '1';
  integer counters = GetNumEventCounters();
  if counters != 0 then
    for idx = 0 to counters - 1
      E = if AArch32.PMUCounterIsHyp(idx) then hpme else PMCR.E;
      if E == '1' && PMINTENSET<idx> == '1' && PMOVSSET<idx> == '1' then pmuirq = TRUE;

  SetInterruptRequestLevel(InterruptID_PMUIRQ, if pmuirq then HIGH else LOW);

  CTI_SetEventLevel(CrossTriggerIn_PMUOverflow, if pmuirq then HIGH else 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 PMOVSCLR.)
```

Library pseudocode for aarch32/debug/pmu/AArch32.ClearEventCounters

```
// AArch32.ClearEventCounters()
// =====
// Zero all the event counters.

AArch32.ClearEventCounters()
  if HaveAArch64() then
    // Force the counter to be cleared as a 64-bit counter.
    AArch64.ClearEventCounters();
    return;

  integer counters = AArch32.GetNumEventCountersAccessible();
  if counters != 0 then
    for idx = 0 to counters - 1
      PMEVCNTR[idx] = Zeros();
```



```

// AArch32.CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch32.CountPMUEvents(integer idx)
    assert idx == CYCLE_COUNTER_ID || idx < GetNumEventCounters();
    if !ELUsingAArch32(EL1) then return AArch64.CountPMUEvents(idx);
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted();

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch32.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE_COUNTER_ID then
        enabled = PMCR.E == '1' && PMCNTENSET.C == '1';
    else
        if resvd_for_el2 then
            E = if ELUsingAArch32(EL2) then HDCR.HPME else MDCR_EL2.HPME;
        else
            E = PMCR.E;
        enabled = E == '1' && PMCNTENSET<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state 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
    // * Not executing at EL0, or SDER.SUNIDEN == 0
    if HaveEL(EL3) && ss == SS_Secure then
        spme = if ELUsingAArch32(EL3) then SDCR.SPME else MDCR_EL3.SPME;
        if !ELUsingAArch32(EL3) && HavePMUv3p7() then
            prohibited = spme == '0' && MDCR_EL3.MPMX == '0';
        else
            prohibited = spme == '0';
        if prohibited && PSTATE.EL == EL0 then
            prohibited = SDER.SUNIDEN == '0';

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * HDCR.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = HDCR.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

    // If FEAT_PMUv3p7 is implemented, event counting can be frozen
    if HavePMUv3p7() then

```

```

bits(5) hpmn;
if HaveEL(EL2) then
    hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
ovflws = ZeroExtend(PMOVSSSET<GetNumEventCounters()-1:0>);
bit FZ;
if resvd_for_el2 then
    FZ = if ELUsingAArch32(EL2) then HDCR.HPMFZO else MDCR_EL2.HPMFZO;
    ovflws<UInt(hpmn)-1:0> = Zeros();
else
    FZ = PMCR.FZO;
    if HaveEL(EL2) && UInt(hpmn) < GetNumEventCounters() then
        ovflws<GetNumEventCounters()-1:UInt(hpmn)> = Zeros();
frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR.DP == '0');
// Otherwise whether event counting is prohibited 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.DP.
if HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
    if HaveEL(EL3) && ss == SS_Secure then
        sccd = if ELUsingAArch32(EL3) then SDCR.SCCD else MDCR_EL3.SCCD;
        if sccd == '1' then prohibited = TRUE;
    if PSTATE.EL == EL2 && HDCR.HCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH} bits
filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR 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';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = NSH == '0';
    when EL3 filtered = P == '1';

return !debug && enabled && !prohibited && !filtered && !frozen;

```

Library pseudocode for aarch32/debug/pmu/AArch32.GetNumEventCountersAccessible

```

// AArch32.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch32.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters();
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled() then
        n = UInt(if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN);
        if n > total_counters || (!HaveFeatHPMN0() && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable_PMUEVENTCOUNTER);
    else
        n = total_counters;

return n;

```

Library pseudocode for aarch32/debug/pmu/AArch32.IncrementEventCounter

```
// AArch32.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch32.IncrementEventCounter(integer idx, integer increment)
  if HaveAArch64() then
    // Force the counter to be incremented as a 64-bit counter.
    AArch64.IncrementEventCounter(idx, increment);
    return;

// 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;
integer ovflw;
bit lp;
old_value = UInt(PMEVCNTR[idx]);
new_value = old_value + PMUCountValue(idx, increment);

PMEVCNTR[idx] = new_value<31:0>;
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 < GetNumEventCounters() then
    PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUCounterIsHyp

```
// AArch32.PMUCounterIsHyp
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch32.PMUCounterIsHyp(integer n)
  boolean resvd_for_el2;
  // Software can reserve some event counters for EL2
  if n != CYCLE_COUNTER_ID && HaveEL(EL2) then
    bits(5) hpmn = if !ELUsingAArch32(EL2) then MDCR_EL2.HPMN else HDCR.HPMN;
    resvd_for_el2 = n >= UInt(hpmn);
    if UInt(hpmn) > GetNumEventCounters() || (!HaveFeatHPMN0() && IsZero(hpmn)) then
      resvd_for_el2 = boolean UNKNOWN;
  else
    resvd_for_el2 = FALSE;

  return resvd_for_el2;
```

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 !HavePMUv3() then
    return;

  PMUEvent(PMU_EVENT_CPU_CYCLES);

  integer counters = GetNumEventCounters();
  if counters != 0 then
    for idx = 0 to counters - 1
      if AArch32.CountPMUEvents(idx) then
        accumulated = PMUEventAccumulator[idx];
        AArch32.IncrementEventCounter(idx, accumulated);
        PMUEventAccumulator[idx] = 0;

  integer old_value;
  integer new_value;
  integer ovflw;
  if (AArch32.CountPMUEvents(CYCLE_COUNTER_ID) &&
      (PMCR.LC == '1' || PMCR.D == '0' || HasElapsed64Cycles())) then
    old_value = UInt(PMCCNTR);
    new_value = old_value + 1;
    PMCCNTR = new_value<63:0>;

    ovflw = if PMCR.LC == '1' then 64 else 32;

    if old_value<64:ovflw> != new_value<64:ovflw> then
      PMOVSSET.C = '1';
      PMOVSRR.C = '1';

  AArch32.CheckForPMUOverflow();
```

Library pseudocode for aarch32/debug/pmu/AArch32.PMUSwIncrement

```
// AArch32.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC.

AArch32.PMUSwIncrement(bits(32) sw_incr)
  integer counters = AArch32.GetNumEventCountersAccessible();
  if counters != 0 then
    for idx = 0 to counters - 1
      if sw_incr<idx> == '1' then
        PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

Library pseudocode for aarch32/debug/takeexceptiondbg/AArch32.EnterHypModeInDebugState

```
// AArch32.EnterHypModeInDebugState()
// =====
// Take an exception in Debug state to Hyp mode.

AArch32.EnterHypModeInDebugState(ExceptionRecord exception)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = 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;
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() 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[] = 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;
    PSTATE.E = SCTL.R.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() 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[] = 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 HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    DLR = bits(32) UNKNOWN;
    DSPSR = 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)

integer top = 31;
bottom = if DBGWVR[n]<2> == '1' then 2 else 3;          // Word or doubleword
byte_select_match = (DBGWCR[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR[n].MASK);

// If DBGWCR[n].MASK is non-zero 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

boolean WVR_match;
if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
            top = 63;
    WVR_match = (vaddress<top:mask> == DBGWVR[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR[n]<mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable_WPMASKEDBITS);
else
    WVR_match = vaddress<top:bottom> == DBGWVR[n]<top:bottom>;

return WVR_match && byte_select_match;
```

Library pseudocode for aarch32/debug/watchpoint/AArch32.WatchpointMatch

```
// AArch32.WatchpointMatch()
// =====
// Watchpoint matching in an AArch32 translation regime.

boolean AArch32.WatchpointMatch(integer n, bits(32) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    // * FALSE for all loads, stores, and atomic operations executed at EL0.
    // * FALSE if the access is unprivileged.
    // * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR[n].E == '1';
    linked = DBGWCR[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch32.StateMatch(DBGWCR[n].SSC, DBGWCR[n].HMC, DBGWCR[n].PAC,
                                     linked, DBGWCR[n].LBN, isbreakpnt, ispriv);
    ls_match = FALSE;
    if iswrite then
        ls_match = (DBGWCR[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR[n].LSC<0> != '0');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch32.WatchpointByteMatch(n, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.Abort

```
// AArch32.Abort()
// =====
// Abort and Debug exception handling in an AArch32 translation regime.

AArch32.Abort(bits(32) vaddress, FaultRecord fault)

    // 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' || IsSecondStage(fault) ||
                            (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
                            (IsDebugException(fault) && MDCR_EL2.TDE == '1'));

    if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
        route_to_aarch64 = SCR_EL3.EA == '1' && IsExternalAbort(fault);

    if route_to_aarch64 then
        AArch64.Abort(ZeroExtend(vaddress), fault);
    elseif fault.acctype == AccType_IFETCH then
        AArch32.TakePrefetchAbortException(vaddress, fault);
    else
        AArch32.TakeDataAbortException(vaddress, 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(32) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype == Exception_DataAbort;

    exception.syndrome = AArch32.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = ZeroExtend(fault.ipaddress.address);
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.CheckPCAlignment

```
// AArch32.CheckPCAlignment()
// =====

AArch32.CheckPCAlignment()

    bits(32) pc = ThisInstrAddr();
    if (CurrentInstrSet() == InstrSet_A32 && pc<1> == '1') || pc<0> == '1' then
        if AArch32.GeneralExceptionsToAArch64() then AArch64.PCAlignmentFault();

    // Generate an Alignment fault Prefetch Abort exception
    vaddress = pc;
    acctype = AccType_IFETCH;
    iswrite = FALSE;
    secondstage = FALSE;
    AArch32.Abort(vaddress, AlignmentFault(acctype, iswrite, secondstage));
```

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, bits(32) vaddress)
    long_format = FALSE;
    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 "Stage 2 synchronous External"))))
    else
        long_format = TTBCR.EAE == '1';
    d_side = TRUE;
    bits(32) syndrome;
    if long_format then
        syndrome = AArch32.FaultStatusLD(d_side, fault);
    else
        syndrome = AArch32.FaultStatusSD(d_side, fault);

    if fault.acctype == AccType\_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 = vaddress;
    else
        DFSR = syndrome;
        DFAR = vaddress;

    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, bits(32) vaddress)
// 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';

d_side = FALSE;
bits(32) fsr;
if long_format then
    fsr = AArch32.FaultStatusLD(d_side, fault);
else
    fsr = AArch32.FaultStatusSD(d_side, fault);

if route_to_monitor then
    IFSR_S = fsr;
    IFAR_S = vaddress;
else
    IFSR = fsr;
    IFAR = vaddress;

return;
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakeDataAbortException

```
// AArch32.TakeDataAbortException()
// =====

AArch32.TakeDataAbortException(bits(32) vaddress, FaultRecord fault)
route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
    (HCR.TGE == '1' ||
    (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
    (IsDebugException(fault) && HDCR.TDE == '1') ||
    IsSecondStage(fault)));

bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

if IsDebugException(fault) then DBGDSCRExt.MOE = fault.debugmoe;
if route_to_monitor then
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
elseif PSTATE.EL == EL2 || route_to_hyp then
    exception = AArch32.AbortSyndrome(Exception\_DataAbort, fault, vaddress);
    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
    else
        AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
else
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    AArch32.EnterMode(M32\_Abort, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/aborts/AArch32.TakePrefetchAbortException

```
// AArch32.TakePrefetchAbortException()
// =====

AArch32.TakePrefetchAbortException(bits(32) vaddress, FaultRecord fault)
    route_to_monitor = HaveEL(EL3) && SCR.EA == '1' && IsExternalAbort(fault);
    route_to_hyp = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR.TGE == '1' ||
        (HaveRASExt() && HCR2.TEA == '1' && IsExternalAbort(fault)) ||
        (IsDebugException(fault) && HDCR.TDE == '1') ||
        IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(32) preferred_exception_return = ThisInstrAddr();
    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, vaddress);
        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
            exception = ExceptionSyndrome(Exception_PCAalignment);
            exception.vaddress = ThisInstrAddr();
        else
            exception = AArch32.AbortSyndrome(Exception_InstructionAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    else
        AArch32.ReportPrefetchAbort(route_to_monitor, fault, vaddress);
        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'));
    bits(32) preferred_exception_return = ThisInstrAddr();
    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
        exception = ExceptionSyndrome(Exception_FIQ);
        AArch32.EnterHypMode(exception, 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'));
bits(32) preferred_exception_return = ThisInstrAddr();
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
    exception = ExceptionSyndrome(Exception_IRQ);
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
else
    AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakePhysicalSErrorException

```
// AArch32.TakePhysicalSErrorException()
// =====

AArch32.TakePhysicalSErrorException(boolean parity, bit extflag, bits(2) pe_error_state,
                                     bits(25) full_syndrome)

// 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' || (!IsInHost() && HCR_EL2.AMO == '1'));
if !route_to_aarch64 && HaveEL(EL3) && !ELUsingAArch32(EL3) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.TakePhysicalSErrorException(full_syndrome);

route_to_monitor = HaveEL(EL3) && SCR.EA == '1';
route_to_hyp = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
               (HCR.TGE == '1' || HCR.AMO == '1'));
bits(32) preferred_exception_return = ThisInstrAddr();
vect_offset = 0x10;
lr_offset = 8;

bits(2) target_el;
if route_to_monitor then
    target_el = EL3;
elseif PSTATE.EL == EL2 || route_to_hyp then
    target_el = EL2;
else
    target_el = EL1;

if IsSErrorEdgeTriggered(target_el, full_syndrome) then
    ClearPendingPhysicalSError();

fault = AsyncExternalAbort(parity, pe_error_state, extflag);
vaddress = bits(32) UNKNOWN;

case target_el of
    when EL3
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
    when EL2
        exception = AArch32.AbortSyndrome(Exception_DataAbort, fault, vaddress);
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
    when EL1
        AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
        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 FM0==1
    assert HCR.TGE == '0' && HCR.FM0 == '1';
  else
    assert HCR_EL2.TGE == '0' && HCR_EL2.FM0 == '1';
  // Check if routed to AArch64 state
  if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualFIQException();

  bits(32) preferred_exception_return = ThisInstrAddr();
  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 IM0==1
    assert HCR.TGE == '0' && HCR.IM0 == '1';
  else
    assert HCR_EL2.TGE == '0' && HCR_EL2.IM0 == '1';

  // Check if routed to AArch64 state
  if PSTATE.EL == EL0 && !ELUsingAArch32(EL1) then AArch64.TakeVirtualIRQException();

  bits(32) preferred_exception_return = ThisInstrAddr();
  vect_offset = 0x18;
  lr_offset = 4;

  AArch32.EnterMode(M32_IRQ, preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/async/AArch32.TakeVirtualErrorException

```
// AArch32.TakeVirtualErrorException()
// =====

AArch32.TakeVirtualErrorException(bit extflag, bits(2) pe_error_state, bits(25) full_syndrome)

    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.TakeVirtualErrorException(full_syndrome);

    route_to_monitor = FALSE;

    bits(32) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x10;
    lr_offset = 8;

    vaddress = bits(32) UNKNOWN;
    parity = FALSE;
    FaultRecord fault;
    if HaveRASExt() then
        if ELUsingAArch32(EL2) then
            fault = AsyncExternalAbort(FALSE, VDFSR.AET, VDFSR.ExT);
        else
            fault = AsyncExternalAbort(FALSE, VESR_EL2.AET, VESR_EL2.ExT);
    else
        fault = AsyncExternalAbort(parity, pe_error_state, extflag);

    ClearPendingVirtualSError();
    AArch32.ReportDataAbort(route_to_monitor, fault, vaddress);
    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);
    vaddress = bits(32) UNKNOWN;
    acctype = AccType_IFETCH; // Take as a Prefetch Abort
    iswrite = FALSE;
    entry = DebugException_BKPT;

    fault = AArch32.DebugFault(acctype, iswrite, entry);
    AArch32.Abort(vaddress, fault);
```

Library pseudocode for aarch32/exceptions/debug/DebugException

```
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') // FPSID
        || (tid3 == '1' && reg IN {'0101', '0110', '0111'}) then // MVFRx
        if ELUsingAArch32(EL2) then
            AArch32.SystemAccessTrap(M32_Hyp, 0x8); // Exception_AdvSIMDFPAccessTrap
        else
            AArch64.AArch32SystemAccessTrap(EL2, 0x8); // Exception_AdvSIMDFPAccessTrap
```

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_CP14RRTTrap        ec = 0x05;
        when Exception_CP14DTTrap         ec = 0x06;
        when Exception_AdvSIMDFPAccessTrap ec = 0x07;
        when Exception_FPIDTrap           ec = 0x08;
        when Exception_PACTrap            ec = 0x09;
        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;
        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 exception)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch32.ExceptionClass(exceptype);
    iss = exception.syndrome;

    // 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_PCAlignment} then
        HIFAR = exception.vaddress<31:0>;
        HDFAR = bits(32) UNKNOWN;
    elseif exceptype == Exception_DataAbort then
        HIFAR = bits(32) UNKNOWN;
        HDFAR = exception.vaddress<31:0>;

    if exception.ipavalid then
        HPFAR<31:4> = exception.ipaddress<39:12>;
    else
        HPFAR<31:4> = bits(28) UNKNOWN;

    return;
```

Library pseudocode for aarch32/exceptions/exceptions/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();
    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');

    boolean branch_conditional = FALSE;
    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,I0F
  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.TakeHVCEXception

```
// AArch32.TakeHVCEXception()
// =====

AArch32.TakeHVCEXception(bits(16) immediate)
  assert HaveEL\(EL2\) && ELUsingAArch32\(EL2\);

  AArch32.ITAdvance();
  SSAdvance();
  bits(32) preferred_exception_return = NextInstrAddr();
  vect_offset = 0x08;

  exception = ExceptionSyndrome\(Exception\_HypervisorCall\);
  exception.syndrome<15:0> = immediate;

  if PSTATE.EL == EL2 then
    AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
  else
    AArch32.EnterHypMode(exception, preferred_exception_return, 0x14);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSMCException

```
// AArch32.TakeSMCException()
// =====

AArch32.TakeSMCException()
    assert HaveEL(EL3) && ELUsingAArch32(EL3);
    AArch32.ITAdvance();
    SSAdvance();
    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    AArch32.EnterMonitorMode(preferred_exception_return, lr_offset, vect_offset);
```

Library pseudocode for aarch32/exceptions/syscalls/AArch32.TakeSVCException

```
// AArch32.TakeSVCException()
// =====

AArch32.TakeSVCException(bits(16) immediate)

    AArch32.ITAdvance();
    SSAdvance();
    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

    bits(32) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x08;
    lr_offset = 0;

    if PSTATE.EL == EL2 || route_to_hyp then
        exception = ExceptionSyndrome(Exception_SupervisorCall);
        exception.syndrome<15:0> = immediate;
        if PSTATE.EL == EL2 then
            AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
        else
            AArch32.EnterHypMode(exception, 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 exception, bits(32) preferred_exception_return,
                    integer vect_offset)
    SynchronizeContext();
    assert HaveEL(EL2) && CurrentSecurityState() == SS_NonSecure && ELUsingAArch32(EL2);

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
        AArch32.ReportHypEntry(exception);
    AArch32.WriteMode(M32_Hyp);
    SPSR[] = spsr;
    ELR_hyp = preferred_exception_return;
    PSTATE.T = HSCTLR.TE; // PSTATE.J is RES0
    PSTATE.SS = '0';
    if !HaveEL(EL3) || SCR_GEN[].EA == '0' then PSTATE.A = '1';
    if !HaveEL(EL3) || SCR_GEN[].IRQ == '0' then PSTATE.I = '1';
    if !HaveEL(EL3) || SCR_GEN[].FIQ == '0' then PSTATE.F = '1';
    PSTATE.E = HSCTLR.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HaveSSBSExt() then PSTATE.SSBS = HSCTLR.DSSBS;
    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;

    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(target_mode);
    SPSR[] = 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 HavePANExt() && SCTL.R.SPAN == '0' then PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTL.R.DSSBS;
    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;
    bits(32) spsr = GetPSRFromPSTATE(AArch32_NonDebugState);
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    SPSR[] = spsr;
    R[14] = preferred_exception_return + lr_offset;
    PSTATE.T = SCTLRT.E; // PSTATE.J is RES0
    PSTATE.SS = '0';
    PSTATE.<A,I,F> = '111';
    PSTATE.E = SCTLRE.EE;
    PSTATE.IL = '0';
    PSTATE.IT = '00000000';
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTLRT.SPAN == '0' then
            PSTATE.PAN = '1';
    if HaveSSBSExt() then PSTATE.SSBS = SCTLRT.DSSBS;
    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, boolean advsimd)
  if PSTATE.EL == EL0 && (!EL2Enabled() || (!ELUsingAArch32(EL2) && HCR_EL2.TGE == '0')) && !ELUsingAArch32(EL2)
    // The PE behaves as if FPEXC.EN is 1
    AArch64.CheckFPEEnabled();
    AArch64.CheckFPAdvSIMDEnabled();
  elsif PSTATE.EL == EL0 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' && !ELUsingAArch32(EL2)
    if fpexc_check && HCR_EL2.RW == '0' then
      fpexc_en = bits(1) IMPLEMENTATION_DEFINED "FPEXC.EN value when TGE==1 and RW==0";
      if fpexc_en == '0' then UNDEFINED;
      AArch64.CheckFPEEnabled();
    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 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 UNDEFINED;

      // If required, check FPEXC enabled bit.
      if fpexc_check && FPEXC.EN == '0' then 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' && EL3SDDTrapPriority\(\)) 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
          exception = ExceptionSyndrome\(Exception\_AdvSIMDFPAccessTrap\);
          exception.syndrome<24:20> = ConditionSyndrome\(\);

          if advsimd then
            exception.syndrome<5> = '1';
          else
            exception.syndrome<5> = '0';
            exception.syndrome<3:0> = '1010';           // coproc field, always 0xA

          if PSTATE.EL == EL2 then
            AArch32.TakeUndefInstrException(exception);
          else
            AArch32.TakeHypTrapException(exception);

    if HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
      // Check if access disabled in CPTR_EL3
      if CPTR_EL3.TFP == '1' then
        if Halted\(\) && EDSCR.SDD == '1' 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
      exception = ExceptionSyndrome\(Exception\_MonitorCall\);
      AArch32.TakeHypTrapException(exception);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.CheckForSVCTrap

```
// AArch32.CheckForSVCTrap()
// =====
// Check for trap on SVC instruction

AArch32.CheckForSVCTrap(bits(16) immediate)
  if HaveFGTExt() then
    route_to_el2 = FALSE;
    if PSTATE.EL == EL0 then
      route_to_el2 = (!ELUsingAArch32(EL1) && EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
        (HCR_EL2.<E2H, TGE> != '11' && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1')));

    if route_to_el2 then
      exception = ExceptionSyndrome(Exception_SupervisorCall);
      exception.syndrome<15:0> = immediate;
      exception.trappedsyscallinst = TRUE;
      bits(64) preferred_exception_return = ThisInstrAddr();
      vect_offset = 0x0;

      AArch64.TakeException(EL2, exception, 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
    AArch64.CheckForWFXTrap(target_el, wfxtype);
    return;

  boolean is_wfe = wfxtype == WFXType_WFE;
  boolean trap;
  case target_el of
    when EL1
      trap = (if is_wfe then SCTLR.nTWE else SCTLR.nTWI) == '0';
    when EL2
      trap = (if is_wfe then HCR.TWE else HCR.TWI) == '1';
    when EL3
      trap = (if is_wfe then SCR.TWE else SCR.TWI) == '1';

  if trap then
    if target_el == EL1 && EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.TGE == '1' then
      AArch64.WFXTrap(wfxtype, target_el);

    if target_el == EL3 then
      AArch32.TakeMonitorTrapException();
    elseif target_el == EL2 then
      exception = ExceptionSyndrome(Exception_WFXTrap);
      exception.syndrome<24:20> = ConditionSyndrome();

      case wfxtype of
        when WFXType_WFI
          exception.syndrome<0> = '0';
        when WFXType_WFE
          exception.syndrome<0> = '1';

      AArch32.TakeHypTrapException(exception);
    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 SCTL.R.ITD else SCTL.R[.].ITD);
    if it_disabled == '1' then
        if mask != '1000' then UNDEFINED;

    // Otherwise whether the IT block is allowed depends on hw1 of the next instruction.
    next_instr = AArch32.MemSingle[NextInstrAddr(), 2, AccType_IFETCH, TRUE];

    if next_instr IN {'11xxxxxxxxxxxx', '1011xxxxxxxxxxxx', '10100xxxxxxxxxxxx',
                    '01001xxxxxxxxxxxx', '010001xxx1111xxx', '010001xx1xxxx111'} then
        // It is IMPLEMENTATION DEFINED whether the Undefined Instruction exception is
        // taken on the IT instruction or the next instruction. This is not reflected in
        // the pseudocode, which always takes the exception on the IT instruction. This
        // also does not take into account cases where the next instruction is UNPREDICTABLE.
        UNDEFINED;

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();
    elsif PSTATE.IL == '1' then
        route_to_hyp = PSTATE.EL == EL0 && EL2Enabled() && HCR.TGE == '1';

        bits(32) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x04;

        if PSTATE.EL == EL2 || route_to_hyp then
            exception = ExceptionSyndrome(Exception_IllegalState);
            if PSTATE.EL == EL2 then
                AArch32.EnterHypMode(exception, preferred_exception_return, vect_offset);
            else
                AArch32.EnterHypMode(exception, 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 SCTL.R.SED else SCTL.R[.].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
        exception = AArch32.SystemAccessTrapSyndrome(ThisInstr(), ec);
        AArch32.TakeHypTrapException(exception);
    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 exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elsif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RTTTrap} 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 exception.exceptype != Exception_FPIDTrap then // When trap is not for VMRS
            iss<19:17> = instr<7:5>;         // opc2
            iss<16:14> = instr<23:21>;       // opc1
            iss<4:1>   = instr<3:0>;         //CRm
        else //VMRS Access
            iss<19:17> = '000';              //opc2 - Hardcoded for VMRS
            iss<16:14> = '111';              //opc1 - Hardcoded for VMRS
            iss<4:1>   = '0000';            //CRm - Hardcoded for VMRS
        elsif exception.exceptype IN {Exception_CP14RRTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTrap} then
            // Trapped MRRC/MCRR, VMRS/VMSR
            iss<19:16> = instr<7:4>;         // opc1
            iss<13:10> = instr<19:16>;       // Rt2
            iss<8:5>  = instr<15:12>;       // Rt
            iss<4:1>  = instr<3:0>;         // CRm
        elsif exception.exceptype == Exception_CP14DTTTrap 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

        exception.syndrome<24:20> = ConditionSyndrome();
        exception.syndrome<19:0>  = iss;

    return exception;
```

Library pseudocode for aarch32/exceptions/traps/AArch32.TakeHypTrapException

```
// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException(integer ec)
    exception = AArch32.SystemAccessTrapSyndrome\(ThisInstr\(\), ec\);
    AArch32.TakeHypTrapException\(exception\);

// AArch32.TakeHypTrapException()
// =====
// Exceptions routed to Hyp mode as a Hyp Trap exception.

AArch32.TakeHypTrapException\(ExceptionRecord exception\)
    assert HaveEL\(EL2\) && CurrentSecurityState\(\) == SS\_NonSecure && ELUsingAArch32\(EL2\);

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x14;

    AArch32.EnterHypMode\(exception, preferred\_exception\_return, vect\_offset\);
```

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\(EL3\);

    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    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()
    exception = ExceptionSyndrome\(Exception\_Uncategorized\);
    AArch32.TakeUndefInstrException\(exception\);

// AArch32.TakeUndefInstrException()
// =====

AArch32.TakeUndefInstrException(ExceptionRecord exception)

    route_to_hyp = PSTATE.EL == EL0 && EL2Enabled\(\) && HCR.TGE == '1';
    bits(32) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x04;
    lr_offset = if CurrentInstrSet\(\) == InstrSet\_A32 then 4 else 2;

    if PSTATE.EL == EL2 then
        AArch32.EnterHypMode\(exception, preferred\_exception\_return, vect\_offset\);
    elsif route_to_hyp then
        AArch32.EnterHypMode\(exception, preferred\_exception\_return, 0x14\);
    else
        AArch32.EnterMode\(M32\_Undef, preferred\_exception\_return, lr\_offset, vect\_offset\);
```

Library pseudocode for aarch32/exceptions/traps/AArch32.UndefinedFault

```
// AArch32.UndefinedFault()
// =====
AArch32.UndefinedFault()
    if AArch32.GeneralExceptionsToAArch64\(\) then AArch64.UndefinedFault\(\);
    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.FaultStatusLD

```
// AArch32.FaultStatusLD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Long-descriptor format.
bits(32) AArch32.FaultStatusLD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;
    bits(32) fsr = Zeros\(\);
    if HaveRASExt\(\) && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC,
            AccType\_AT, AccType\_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '1';
    fsr<5:0> = EncodeLDFSC(fault.statuscode, fault.level);
    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultStatusSD

```
// AArch32.FaultStatusSD()
// =====
// Creates an exception fault status value for Abort and Watchpoint exceptions taken
// to Abort mode using AArch32 and Short-descriptor format.

bits(32) AArch32.FaultStatusSD(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(32) fsr = Zeros();
    if HaveRASExt() && IsAsyncAbort(fault) then fsr<15:14> = fault.errortype;
    if d_side then
        if fault.acctype IN {AccType\_DC, AccType\_IC,
            AccType\_AT, AccType\_ATPAN} then
            fsr<13> = '1'; fsr<11> = '1';
        else
            fsr<11> = if fault.write then '1' else '0';
    if IsExternalAbort(fault) then fsr<12> = fault.extflag;
    fsr<9> = '0';
    fsr<10,3:0> = EncodeSDFSC(fault.statuscode, fault.level);
    if d_side then
        fsr<7:4> = fault.domain;           // Domain field (data fault only)

    return fsr;
```

Library pseudocode for aarch32/functions/aborts/AArch32.FaultSyndrome

```
// AArch32.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// AArch32 Hyp mode.

bits(25) AArch32.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros();

    if HaveRASExt() && IsAsyncAbort(fault) then
        iss<11:10> = fault.errortype; // AET

    if d_side then
        if (IsSecondStage(fault) && !fault.s2fslwalk &&
            (!IsExternalSyncAbort(fault) ||
            (!HaveRASExt() && fault.acctype == AccType\_TTW &&
            boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
            iss<24:14> = LSInstructionSyndrome();

        if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT, AccType\_ATPAN} then
            iss<8> = '1'; iss<6> = '1';
        else
            iss<6> = if fault.write then '1' else '0';

    if IsExternalAbort(fault) then iss<9> = fault.extflag;
    iss<7> = if fault.s2fslwalk then '1' else '0';
    iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return 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

```
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

```
enumeration VBitOps {VBitOps_VBIF, VBitOps_VBIT, VBitOps_VBSL};
```

Library pseudocode for aarch32/functions/common/VCGEType

```
enumeration VCGEType {VCGEType_signed, VCGEType_unsigned, VCGEType_fp};
```

Library pseudocode for aarch32/functions/common/VCGTtype

```
enumeration VCGTtype {VCGTtype_signed, VCGTtype_unsigned, VCGTtype_fp};
```

Library pseudocode for aarch32/functions/common/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.

boolean AArch32.ExclusiveMonitorsPass(bits(32) address, integer size)

    // 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.

    acctype = AccType_ATOMIC;
    iswrite = TRUE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch32.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch32/functions/exclusive/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

```
// 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)
    acctype = AccType\_ATOMIC;
    iswrite = FALSE;

    aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        return;

    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        MarkExclusiveGlobal(memaddrdesc.paddress, ProcessorID(), size);

    MarkExclusiveLocal(memaddrdesc.paddress, 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.CheckAdvSIMDOrVFPEEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrVFPEEnabled() 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.CheckAdvSIMDOrVFPEEnabled(include_fpexc_check, advsimd);
  // Return from CheckAdvSIMDOrVFPEEnabled() 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, FPCRTType 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);
      FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
      result = FPInfinity('0');
    elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
      result = FPInfinity('1');
    elsif zero1 && zero2 && sign1 != sign2 then
      result = FPZero(sign1);
    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);
      else
        result = FPRound(result_value, fpcr);
  return result;
```

Library pseudocode for aarch32/functions/float/FPRSqrtStep

```
// FPRSqrtStep()
// =====

bits(N) FPRSqrtStep(bits(N) op1, bits(N) op2)
  assert N IN {16,32};
  FPCRTYPE fpcr = StandardFPSCRValue();
  (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');
    else
      product = FPMul(op1, op2, fpcr);
    bits(N) three = FPTThree('0');
    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};
  FPCRTYPE fpcr = StandardFPSCRValue();
  (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');
    else
      product = FPMul(op1, op2, fpcr);
    bits(N) two = FPTTwo('0');
    result = FPSub(two, product, fpcr);
  return result;
```

Library pseudocode for aarch32/functions/float/StandardFPSCRValue

```
// StandardFPSCRValue()
// =====

FPCRTYPE StandardFPSCRValue()
  bits(32) upper = '00000000000000000000000000000000';
  bits(32) lower = '00000' : FPSCR.AHP : '110000' : FPSCR.FZ16 : '00000000000000000000';
  return upper : lower;
```

Library pseudocode for aarch32/functions/memory/AArch32.CheckAlignment

```
// AArch32.CheckAlignment()
// =====

boolean AArch32.CheckAlignment(bits(32) address, integer alignment, AccType acctype,
                               boolean iswrite)

    bit A;
    if PSTATE.EL == EL0 && !ELUsingAArch32\(S1TranslationRegime\(\)\) then
        A = SCTLRA.A; //use AArch64 register, when higher Exception level is using AArch64
    elsif PSTATE.EL == EL2 then
        A = HSCTLR.A;
    else
        A = SCTLRA.A;
    aligned = (address == Align(address, alignment));
    atomic = acctype IN { AccType\_ATOMIC, AccType\_ATOMICRW, AccType\_ORDEREDATOMIC,
                        AccType\_ORDEREDATOMICRW, AccType\_ATOMICS64, AccType\_A32LSMD};
    ordered = acctype IN { AccType\_ORDERED, AccType\_ORDEREDRW, AccType\_LIMITEDORDERED,
                        AccType\_ORDEREDATOMIC, AccType\_ORDEREDATOMICRW };
    vector = acctype == AccType\_VEC;

    // AccType_VEC is used for SIMD element alignment checks only
    check = (atomic || ordered || vector || A == '1');

    if check && !aligned then
        secondstage = FALSE;
        AArch32.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```



```

// 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, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch32.MemSingle[address, size, acctype, aligned, ispair];

// 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, AccType acctype, boolean aligned, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    bits(size*8) value;
    iswrite = FALSE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
    return value;

// AArch32.MemSingle[] - assignment (write) form
// =====

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
    return;

// AArch32.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch32.MemSingle[bits(32) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bits(size*8) value
    assert size IN {1, 2, 4, 8, 16};
    assert address == Align(address, size);

    AddressDescriptor memaddrdesc;
    iswrite = TRUE;

    memaddrdesc = AArch32.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch32.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    PhysMemRetStatus memstatus;
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
    return;

```

Library pseudocode for aarch32/functions/memory/Hint_PreloadData

```
Hint_PreloadData(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/Hint_PreloadDataForWrite

```
Hint_PreloadDataForWrite(bits(32) address);
```

Library pseudocode for aarch32/functions/memory/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)
    acctype = AccType\_ATOMIC;
    return Mem\_with\_type[address, size, acctype];

// MemA[] - assignment form
// =====

MemA(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_ATOMIC;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemO

```
// MemO[] - non-assignment form
// =====

bits(8*size) MemO(bits(32) address, integer size)
    acctype = AccType\_ORDERED;
    return Mem\_with\_type[address, size, acctype];

// MemO[] - assignment form
// =====

MemO(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_ORDERED;
    Mem\_with\_type[address, size, acctype] = 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)
    acctype = AccType\_A32LSMD;
    return Mem\_with\_type[address, size, acctype];

// MemS[] - assignment form
// =====
// Memory accessor for streaming store multiple instructions

MemS(bits(32) address, integer size) = bits(8*size) value
    acctype = AccType\_A32LSMD;
    Mem\_with\_type[address, size, acctype] = value;
    return;
```

Library pseudocode for aarch32/functions/memory/MemU

```
// MemU[] - non-assignment form
// =====

bits(8*size) MemU[bits(32) address, integer size]
    acctype = AccType\_NORMAL;
    return Mem\_with\_type[address, size, acctype];

// MemU[] - assignment form
// =====

MemU[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_NORMAL;
    Mem\_with\_type[address, size, acctype] = 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]
    acctype = AccType\_UNPRIV;
    return Mem\_with\_type[address, size, acctype];

// MemU_unpriv[] - assignment form
// =====

MemU_unpriv[bits(32) address, integer size] = bits(8*size) value
    acctype = AccType\_UNPRIV;
    Mem\_with\_type[address, size, acctype] = 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, AccType acctype]
    boolean ispair = FALSE;
    return Mem_with_type[address, size, acctype, ispair];

bits(size*8) Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then

        assert size > 1;
        value<7:0> = AArch32.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        c = ConstrainUnpredictable(Unpredictable_DEVPAGE2);
        assert c IN {Constraint_FAULT, Constraint_NONE};
        if c == Constraint_NONE then aligned = TRUE;

        for i = 1 to size-1
            value<8*i+7:8*i> = AArch32.MemSingle[address+i, 1, acctype, aligned];
    else
        value = AArch32.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(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, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem_with_type[address, size, acctype, ispair] = value_in;

Mem_with_type[bits(32) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch32.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch32.CheckAlignment(address, size, acctype, iswrite);

    if !aligned then
        assert size > 1;
        AArch32.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory

```

```

// access will generate an Alignment Fault, as to get this far means the first byte did
// not, so we must be changing to a new translation page.
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, acctype, aligned] = value<8*i+7:8*i>;
else
    AArch32.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

```

Library pseudocode for aarch32/functions/ras/AArch32.ESBOperation

```

// AArch32.ESBOperation()
// =====
// Perform the AArch32 ESB operation for ESB executed in AArch32 state

AArch32.ESBOperation()

// 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.AMO == '1';
if !route_to_aarch64 && HaveEL\(EL3\) && !ELUsingAArch32\(EL3\) then
    route_to_aarch64 = SCR_EL3.EA == '1';

if route_to_aarch64 then
    AArch64.ESBOperation\(\);
    return;

route_to_monitor = HaveEL\(EL3\) && ELUsingAArch32\(EL3\) && SCR.EA == '1';
route_to_hyp = PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) && (HCR.TGE == '1' || HCR.AMO == '1');

bits(5) target;
if route_to_monitor then
    target = M32\_Monitor;
elseif route_to_hyp || PSTATE.M == M32\_Hyp then
    target = M32\_Hyp;
else
    target = M32\_Abort;

boolean mask_active;
if CurrentSecurityState\(\) == SS\_Secure then
    mask_active = TRUE;
elseif target == M32\_Monitor then
    mask_active = SCR.AW == '1' && (!HaveEL\(EL2\) || (HCR.TGE == '0' && HCR.AMO == '0'));
else
    mask_active = target == M32\_Abort || PSTATE.M == M32\_Hyp;

mask_set = PSTATE.A == '1';
(-, el) = ELFromM32\(target\);
intdis = Halted\(\) || ExternalDebugInterruptsDisabled\(el\);
masked = intdis || (mask_active && mask_set);

// Check for a masked Physical SError pending that can be synchronized
// by an Error synchronization event.
if masked && IsSynchronizablePhysicalSErrorPending\(\) then
    syndrome32 = AArch32.PhysicalSErrorSyndrome\(\);
    DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
    ClearPendingPhysicalSError\(\);

return;

```

Library pseudocode for aarch32/functions/ras/AArch32.PhysicalSErrorSyndrome

```

// Return the SError syndrome
AArch32.SErrorSyndrome AArch32.PhysicalSErrorSyndrome();

```

Library pseudocode for aarch32/functions/ras/AArch32.ReportDeferredSError

```
// AArch32.ReportDeferredSError()
// =====
// Return deferred SError syndrome

bits(32) AArch32.ReportDeferredSError(bits(2) AET, bit ExT)
    bits(32) target;
    target<31> = '1'; // A
    syndrome = Zeros(16);
    if PSTATE.EL == EL2 then
        syndrome<11:10> = AET; // AET
        syndrome<9> = ExT; // EA
        syndrome<5:0> = '010001'; // DFSC
    else
        syndrome<15:14> = AET; // AET
        syndrome<12> = ExT; // 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
    if HaveAArch64() then
        target<24:0> = ZeroExtend(syndrome); // Any RES0 fields must be set to zero
    else
        target<15:0> = syndrome;
    return target;
```

Library pseudocode for aarch32/functions/ras/AArch32.SErrorSyndrome

```
type AArch32.SErrorSyndrome is (
    bits(2) AET,
    bit ExT
)
```

Library pseudocode for aarch32/functions/ras/AArch32.vESB0peration

```
// AArch32.vESB0peration()
// =====
// Perform the ESB operation for virtual SError interrupts executed in AArch32 state

AArch32.vESB0peration()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // Check for EL2 using AArch64 state
    if !ELUsingAArch32(EL2) then
        AArch64.vESB0peration();
        return;

    // If physical SError interrupts are routed to Hyp mode, and TGE is not set, then a
    // virtual SError interrupt might be pending
    vSEI_enabled = HCR.TGE == '0' && HCR.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR.VA == '1';
    vintdis = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        HCR.VA = '0'; // Clear pending virtual SError

    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(boolean cold_reset);
```

Library pseudocode for aarch32/functions/registers/ALUExceptionReturn

```
// ALUExceptionReturn()
// =====

ALUExceptionReturn(bits(32) address)
  if PSTATE.EL == EL2 then
    UNDEFINED;
  elsif PSTATE.M IN {M32_User,M32_System} then
    Constraint c = ConstrainUnpredictable(Unpredictable_ALUEXCEPTIONRETURN);
    assert c IN {Constraint_UNDEF, Constraint_NOP};
    case c of
      when Constraint_UNDEF
        UNDEFINED;
      when Constraint_NOP
        EndOfInstruction();
    else
      AArch32.ExceptionReturn(address, SPSR[]);
```

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';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'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';
  boolean branch_conditional = !(AArch32.CurrentCond() IN {'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';
    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;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    return vreg<base+63:base>;

// D[] - assignment form
// =====

D[integer n] = bits(64) value
    assert n >= 0 && n <= 31;
    base = (n MOD 2) * 64;
    bits(128) vreg = V[n DIV 2];
    vreg<base+63:base> = value;
    V[n DIV 2] = 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/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_mode_registers

```
bits(32) SP_mon;
bits(32) LR_mon;
```

Library pseudocode for aarch32/functions/registers/PC

```
// PC - non-assignment form
// =====

bits(32) PC
    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 PC;
```

Library pseudocode for aarch32/functions/registers/Q

```
// Q[] - non-assignment form
// =====

bits(128) Q[integer n]
    assert n >= 0 && n <= 15;
    return V[n];

// Q[] - assignment form
// =====

Q[integer n] = bits(128) value
    assert n >= 0 && n <= 15;
    V[n] = 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);
        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;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    return vreg<base+31:base>;

// S[] - assignment form
// =====

S[integer n] = bits(32) value
    assert n >= 0 && n <= 31;
    base = (n MOD 4) * 32;
    bits(128) vreg = V[n DIV 4];
    vreg<base+31:base> = value;
    V[n DIV 4] = vreg;
    return;
```

Library pseudocode for aarch32/functions/registers/SP

```
// SP - assignment form
// =====

SP = bits(32) value
  R[13] = value;
  return;

// SP - non-assignment form
// =====

bits(32) SP
  return R[13];
```

Library pseudocode for aarch32/functions/registers/_Dclone

```
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

  boolean branch_conditional = !(AArch32.CurrentCond() IN {'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> IN {'101x'});
  else // InstrSet_T32
    return (instr<31:28> IN {'111x'} && (instr<27:24> == '1110' || instr<27:25> == '110') && instr<11:8> IN {'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

```
// 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

```
// 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

```
// 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

```
// 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

```
// 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 HaveSSBSExt() then
            PSTATE.SSBS = value<23>;
        if privileged then
            PSTATE.PAN = value<22>;
        if HaveDITExt() 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

```
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[];

    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[] = 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/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, -) = SignedSat0(i, N);
    return result;
```

Library pseudocode for aarch32/functions/v6simd/UnsignedSat

```
// UnsignedSat()
// =====

bits(N) UnsignedSat(integer i, integer N)
    (result, -) = UnsignedSat0(i, N);
    return result;
```



```

// 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;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    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 || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR.FB == '1') then
            cache.shareability = Shareability_ISH;
        else
            cache.shareability = Shareability_NSH;
        cache.regval = ZeroExtend(regval);
        CACHE_OP(cache);
    else
        assert opscope == CacheOpScope_PoU;

        if EL2Enabled() then
            if PSTATE.EL IN {EL0, EL1} then
                cache.is_vmid_valid = TRUE;
                cache.vmid = VMID[];
            else
                cache.is_vmid_valid = FALSE;
        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);
        cache.translated = need_translate;

        if !need_translate then
            cache.paddress = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;

        wasaligned = TRUE;
        iswrite = FALSE;
        size = 0;
        memaddrdesc = AArch32.TranslateAddress(regval, acctype, iswrite, wasaligned, size);
        if IsFault(memaddrdesc) then
            AArch32.Abort(regval, memaddrdesc.fault);

        cache.paddress = memaddrdesc.paddress;
        CACHE_OP(cache);
    return;

```

Library pseudocode for aarch32/predictionrestrict/RestrictPrediction

```
// RestrictPrediction()
// =====
// Clear all predictions in the context.

AArch32.RestrictPrediction(bits(32) val, RestrictType restriction)

    ExecutionCntxt c;
    target_el      = val<25:24>;

    // If the instruction is executed at an EL lower than the specified
    // level, it is treated as a NOP.
    if UInt(target_el) > UInt(PSTATE.EL) then return;

    bit ns = val<26>;
    ss = TargetSecurityState(ns);

    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[];

            elsif 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[];

            elsif 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\(Unpredictable\_RESTEXCB\);
    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.tagged          = FALSE;  
  
    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.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;
elsif HaveAArch32EL(EL3) then
    prrr = PRRR_NS;
    nmrr = NMRR_NS;
else
    prrr = PRRR;
    nmrr = NMRR;

base = 2 * region;
attrfield = prrr<base+1:base>;

if attrfield == '11' then          // Reserved, maps to allocated value
    (-, attrfield) = ConstrainUnpredictableBits(Unpredictable_RESPRRR);

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
        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.tagged = FALSE;

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(bits(32) vaddress, integer size)
    assert ELUsingAArch32\(S1TranslationRegime\(\)\);
    assert size IN {2,4};

    match = FALSE;
    mismatch = FALSE;

    for i = 0 to NumBreakpointsImplemented\(\) - 1
        (match_i, mismatch_i) = AArch32.BreakpointMatch(i, vaddress, size);
        match = match || match_i;
        mismatch = mismatch || mismatch_i;

    if match && HaltOnBreakpointOrWatchpoint\(\) then
        reason = DebugHalt\_Breakpoint;
        Halt(reason);
    elsif (match || mismatch) then
        acctype = AccType\_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException\_Breakpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

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, AccType acctype, boolean iswrite, integer size)

    FaultRecord fault = NoFault();

    boolean d_side = (IsDataAccess(acctype) || acctype == AccType\_DC);
    boolean i_side = (acctype == AccType\_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(vaddress, size);

    if fault.statuscode == Fault\_None && (generate_exception || halt) then
        if d_side then
            fault = AArch32.CheckWatchpoint(vaddress, acctype, iswrite, size);
        elsif i_side then
            fault = AArch32.CheckBreakpoint(vaddress, size);

    if fault.statuscode == Fault\_None && i_side && !vector_catch_first && generate_exception then
        return AArch32.CheckVectorCatch(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(bits(32) vaddress, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    match = AArch32.VCRMATCH(vaddress);
    if size == 4 && !match && AArch32.VCRMATCH(vaddress + 2) then
        match = ConstrainUnpredictableBool(Unpredictable_VCMATCHHALF);

    if match then
        acctype = AccType_IFETCH;
        iswrite = FALSE;
        debugmoe = DebugException_VectorCatch;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

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(bits(32) vaddress, AccType acctype,
    boolean iswrite, integer size)
    assert ELUsingAArch32(S1TranslationRegime());

    if acctype == AccType_DC then
        if !iswrite then
            return NoFault();
        elsif !(boolean IMPLEMENTATION_DEFINED "DCIMVAC generates watchpoint") then
            return NoFault();
    elsif !IsDataAccess(acctype) then
        return NoFault();

    match = FALSE;
    ispriv = AArch32.AccessUsesEL(acctype) != EL0;

    for i = 0 to NumWatchpointsImplemented() - 1
        if AArch32.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;

    if match && HaltOnBreakpointOrWatchpoint() then
        reason = DebugHalt_Watchpoint;
        EDWAR = ZeroExtend(vaddress);
        Halt(reason);
    elsif match then
        debugmoe = DebugException_Watchpoint;
        return AArch32.DebugFault(acctype, iswrite, debugmoe);
    else
        return NoFault();
```

Library pseudocode for aarch32/translation/faults/AArch32.DebugFault

```
// AArch32.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch32.DebugFault(ArchType acctype, boolean iswrite, bits(4) debugmoe)
    FaultRecord fault;

    fault.statuscode = Fault_Debug;
    fault.acctype    = acctype;
    fault.write      = iswrite;
    fault.debugmoe   = debugmoe;
    fault.secondstage = FALSE;
    fault.s2fslwalk  = FALSE;

    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
    iasize = AArch32.S2IASize(walkparams.t0sz);

    return iasize < 40 && !IsZero(ipa<39:iasize>);
```

Library pseudocode for aarch32/translation/faults/AArch32.S1HasAlignmentFault

```
// AArch32.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S1HasAlignmentFault(ArchType acctype, boolean aligned,
    bit ntlsmid, MemoryAttributes memattrs)
    if acctype == ArchType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == ArchType_A32LSMD && ntlsmid == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == ArchType_DCZVA;
```



```

// AArch32.S1LDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 long-descriptor translation
// violates permissions of target memory

boolean AArch32.S1LDHasPermissionsFault(Regime regime, SecurityState ss, S1TTWParams walkparams,
                                         Permissions perms, MemType memtype, PASpace paspace,
                                         boolean ispriv, AccType acctype, boolean iswrite)

bit r;
bit w;
bit x;
bit pr;
bit pw;
bit ur;
bit 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'); // R0 at PL1 only
        when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 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));

    pan_access = !(acctype IN {AccType\_DC, AccType\_IFETCH, AccType\_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if 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 acctype == AccType\_IFETCH then
    constraint = ConstrainUnpredictable(Unpredictable\_INSTRDEVICE);
    if constraint == Constraint\_FAULT && memtype == MemType\_Device then
        return TRUE;
    else
        return x == '0';
elseif acctype IN {AccType\_IC, AccType\_DC} then
    return FALSE;

```

```
elseif iswrite then
    return w == '0';
else
    return r == '0';
```



```

// AArch32.S1SDHasPermissionsFault()
// =====
// Returns whether an access using stage 1 short-descriptor translation
// violates permissions of target memory

boolean AArch32.S1SDHasPermissionsFault(Regime regime, SecurityState ss, Permissions perms_in,
                                         MemType memtype, PASpace paspace, boolean ispriv,
                                         AccType acctype, boolean iswrite)

    Permissions perms = perms_in;
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    SCTLR_Type sctlr;
    if regime == Regime_EL30 then
        sctlr = SCTLR_S;
    elsif HaveAArch32EL(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, R0 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'); // R0 at PL1 only
            when '110' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 at any PL (deprecated)
            when '111' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 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'); // R0 at PL1 only
                when '11' (pr,pw,ur,uw) = ('1','0','1','0'); // R0 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));

    pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT});
    if HavePANExt() && pan_access then
        pan = PSTATE.PAN AND (ur OR uw);
        pr = pr AND NOT(pan);
        pw = pw AND NOT(pan);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);

    // Prevent execution from Non-secure space by PE in Secure state if SIF is set
    if ss == SS_Secure && paspace == PAS_NonSecure then
        x = x AND NOT(if ELUsingAArch32(EL3) then SCR.SIF else SCR_EL3.SIF);

    if acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite then
        return w == '0';
    else
        return r == '0';

```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasAlignmentFault

```
// AArch32.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch32.S2HasAlignmentFault(AccType acctype, boolean aligned,
                                   MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    return !aligned || acctype == AccType_DCZVA;
```

Library pseudocode for aarch32/translation/faults/AArch32.S2HasPermissionsFault

```
// AArch32.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch32.S2HasPermissionsFault(boolean s2fslwalk, S2TTWParams walkparams,
                                       Permissions perms, MemType memtype,
                                       boolean ispriv, AccType acctype,
                                       boolean iswrite)

    bit px;
    bit ux;
    r = perms.s2ap<0>;
    w = perms.s2ap<1>;
    bit x;
    if HaveExtendedExecuteNeverExt() 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 ispriv then px else ux;
    else
        x = r AND NOT(perms.s2xn);

    if s2fslwalk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        else
            return x == '0';
    elsif acctype IN {AccType_IC, AccType_DC} then
        return FALSE;
    elsif iswrite 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
        iasize = AArch32.S1IASize(walkparams.t0sz);
        return walkparams.t0sz != '000' && !IsZero(va<31:iasize>);
    elseif walkparams.t1sz != '000' && walkparams.t0sz != '000' then
        // Lower range Input Address size
        lo_iasize = AArch32.S1IASize(walkparams.t0sz);
        // Upper range Input Address size
        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);
    tlbcontext.tg         = TGx_4KB;
    tlbcontext.includes_s1 = FALSE;
    tlbcontext.includes_s2 = TRUE;
    tlbcontext.ia         = ZeroExtend(ipa.address);
    tlbcontext.cnp        = if HaveCommonNotPrivateTransExt() 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 HaveAArch32EL(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);

    if ttbcr.EAE == '1' then
        tlbcontext.asid = ZeroExtend(if ttbcr.A1 == '0' then ttbr0.ASID else ttbr1.ASID);
    else
        tlbcontext.asid = ZeroExtend(contextidr.ASID);

    tlbcontext.tg = TGx\_4KB;
    tlbcontext.ia = ZeroExtend(va);

    if HaveCommonNotPrivateTransExt() && 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);
    tlbcontext.tg      = TGx\_4KB;
    tlbcontext.cnp     = if HaveCommonNotPrivateTransExt() 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);
    else
        tlbcontext.asid = ZeroExtend(CONTEXTIDR_S.ASID);

    tlbcontext.tg = TGx_4KB;
    tlbcontext.ia = ZeroExtend(va);

    if HaveCommonNotPrivateTransExt() && 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.AccessUsesEL

```
// AArch32.AccessUsesEL()
// =====
// Determine the privilege associated with the access

bits(2) AArch32.AccessUsesEL(AccType acctype)
    if acctype == AccType_UNPRIV then
        return EL0;
    else
        return PSTATE.EL;
```

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) && HaveSecureEL2Ext()) 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, AccType acctype,
                                       boolean iswrite, boolean aligned)

// Prepare fault fields in case a fault is detected
fault = NoFault();
fault.acctype = acctype;
fault.write = iswrite;

regime = TranslationRegime(PSTATE.EL, acctype);
ispriv = PSTATE.EL != EL0 && acctype != AccType_UNPRIV;
ss = SecurityStateForRegime(regime);

// First Stage Translation
AddressDescriptor ipa;
if regime == Regime_EL2 || TTBCR.EAE == '1' then
    (fault, ipa) = AArch32.S1TranslateLD(fault, regime, ss, va, acctype,
                                       aligned, iswrite, ispriv);
else
    (fault, ipa, -) = AArch32.S1TranslateSD(fault, regime, ss, va, acctype,
                                       aligned, iswrite, ispriv);

if fault.statuscode != Fault_None then
    return CreateFaultyAddressDescriptor(ZeroExtend(va), fault);

if regime == Regime_EL10 && EL2Enabled() then
    ipa.vaddress = ZeroExtend(va);
    s2fslwalk = FALSE;
    AddressDescriptor pa;
    (fault, pa) = AArch32.S2Translate(fault, ipa, ss, s2fslwalk, acctype,
                                    aligned, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return CreateFaultyAddressDescriptor(ZeroExtend(va), 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;
    index = 2 * UInt(domain);
    if regime == Regime_EL30 then
        Dn = DACR_S<index+1:index>;
    elsif HaveAArch32EL(EL3) then
        Dn = DACR_NS<index+1:index>;
    else
        Dn = DACR<index+1:index>;

    if Dn == '10' then
        // Reserved value maps to an allocated value
        (-, Dn) = ConstrainUnpredictableBits(Unpredictable_RESDACR);

    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,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned)

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(ss) then
    if ELStateUsingAArch32(EL2, 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 !ELStateUsingAArch32(EL2, ss == SS_Secure) && HaveMTE2Ext() then
        memattrs.tagged = HCR_EL2.DCT == '1';
    else
        memattrs.tagged = FALSE;
elseif acctype == AccType_IFETCH then
    memattrs.memtype = MemType_Normal;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;
    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;
else
    // Treat memory region as Device
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.tagged = FALSE;

bit ntlsmid;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime_EL30 ntlsmid = SCTL_R_S.nTLSMD;
        when Regime_EL2 ntlsmid = HSCTL_R.nTLSMD;
        when Regime_EL10 ntlsmid = if HaveAArch32EL(EL3) then SCTL_R_NS.nTLSMD else SCTL_R.nTLSMD;
else
    ntlsmid = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmid, memattrs) then
    fault.statuscode = Fault_Alignment;
    return (fault, AddressDescriptor UNKNOWN);

FullAddress oa;
oa.address = ZeroExtend(va);
oa.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
ipa = CreateAddressDescriptor(ZeroExtend(va), 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 SCTL_S.M == '1';
    elsif !AArch32.EL2Enabled(ss) then
        return (if HaveAArch32EL(EL3) then SCTL_NS.M else SCTL.M) == '1';
    elsif ELStateUsingAArch32(EL2, ss == SS_Secure) then
        return HCR.<TGE,DC> == '00' && (if HaveAArch32EL(EL3) then SCTL_NS.M else SCTL.M) == '1';
    else
        return HCR_EL2.<TGE,DC> == '00' && SCTL.M == '1';
```



```

// 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,
SecurityState ss, bits(32) va,
AccType acctype, boolean aligned,
boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    return AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

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, ss, walkparams, va, ispriv);

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

SetInGuardedPage(FALSE); // AArch32-VMSA does not guard any pages

if AArch32.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid, walkstate.memattrs) then
    fault.statuscode = Fault\_Alignment;
elseif IsAtomicRW(acctype) then
    if AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
        walkstate.permissions,
        walkstate.memattrs.memtype,
        walkstate.baseaddress.paspace,
        ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault\_Permission;
        fault.write = FALSE;
    elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
        walkstate.permissions,
        walkstate.memattrs.memtype,
        walkstate.baseaddress.paspace,
        ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        fault.statuscode = Fault\_Permission;
        fault.write = TRUE;
elseif AArch32.S1LDHasPermissionsFault(regime, ss, walkparams,
    walkstate.permissions,
    walkstate.memattrs.memtype,
    walkstate.baseaddress.paspace,
    ispriv, acctype, iswrite) then
    fault.statuscode = Fault\_Permission;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if ((acctype == AccType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (acctype != AccType\_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(ss) &&
    (if ELStateUsingAArch32(EL2, 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 = Stage0A(ZeroExtend(va), walkparams.tgx, walkstate);
ipa = CreateAddressDescriptor(ZeroExtend(va), 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,
                                                               SecurityState ss, bits(32) va,
                                                               AccType acctype, boolean aligned,
                                                               boolean iswrite, boolean ispriv)

FaultRecord fault = fault_in;
fault.secondstage = FALSE;
fault.s2fslwalk = FALSE;

if !AArch32.S1Enabled(regime, ss) then
    AddressDescriptor ipa;
    (fault, ipa) = AArch32.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);
    return (fault, ipa, SDType UNKNOWN);

TTWState walkstate;
(fault, walkstate) = AArch32.S1WalkSD(fault, regime, ss, va, ispriv);

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 ntlsmid;
if HaveTrapLoadStoreMultipleDeviceExt() then
    case regime of
        when Regime_EL30 ntlsmid = SCTLR_S.nTLSMD;
        when Regime_EL10 ntlsmid = if HaveAArch32EL(EL3) then SCTLR_NS.nTLSMD else SCTLR.nTLSMD;
else
    ntlsmid = '1';

if AArch32.S1HasAlignmentFault(acctype, aligned, ntlsmid, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif !(acctype IN {AccType_IC, AccType_DC}) && domain == Domain_NoAccess then
    fault.statuscode = Fault_Domain;
elseif domain == Domain_Client then
    if IsAtomicRW(acctype) then
        if AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                            walkstate.memattrs.memtype,
                                            walkstate.baseaddress.paspace,
                                            ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                                walkstate.memattrs.memtype,
                                                walkstate.baseaddress.paspace,
                                                ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
        elseif AArch32.S1SDHasPermissionsFault(regime, ss, walkstate.permissions,
                                                walkstate.memattrs.memtype,
                                                walkstate.baseaddress.paspace,
                                                ispriv, acctype, iswrite) then
            fault.statuscode = Fault_Permission;

if fault.statuscode != Fault_None then
    fault.domain = walkstate.domain;
    return (fault, AddressDescriptor UNKNOWN, walkstate.sdftype);

MemoryAttributes memattrs;
if ((acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || !AArch32.S1ICacheEnabled(regime))) ||
    (acctype != AccType_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(ss) &&
    (if ELStateUsingAArch32(EL2, 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.SDStage0A(walkstate.baseaddress, va, walkstate.sdftype);
ipa = CreateAddressDescriptor(ZeroExtend(va), oa, memattrs);

return (fault, ipa, walkstate.sdftype);

```



```

// AArch32.S2Translate()
// =====
// Perform a stage 2 translation mapping an IPA to a PA

(FaultRecord, AddressDescriptor) AArch32.S2Translate(FaultRecord fault_in, AddressDescriptor ipa,
                                                    SecurityState ss, boolean s2fslwalk,
                                                    AccType acctype, boolean aligned,
                                                    boolean iswrite, boolean ispriv)

    FaultRecord fault = fault_in;
    assert IsZero(ipa.paddress.address<51:40>);

    if !ELStateUsingAArch32(EL2, ss == SS_Secure) then
        slaarch64 = FALSE;
        return AArch64.S2Translate(fault, ipa, slaarch64, ss, s2fslwalk, acctype,
                                    aligned, iswrite, ispriv);

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None;
fault.secondstage = TRUE;
fault.s2fslwalk = s2fslwalk;
fault.ipaddress = ipa.paddress;

walkparams = AArch32.GetS2TTPParams();

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, ipa);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if AArch32.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    assert !s2fslwalk; // AArch32 does not support HW update of TT
    if AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = TRUE;
    elseif AArch32.S2HasPermissionsFault(s2fslwalk, walkparams,
                                    walkstate.permissions,
                                    walkstate.memattrs.memtype,
                                    ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;
MemoryAttributes s2_memattrs;
if ((s2fslwalk &&
    walkstate.memattrs.memtype == MemType_Device) ||
    (acctype == AccType_IFETCH &&
    (walkstate.memattrs.memtype == MemType_Device || HCR2.ID == '1')) ||
    (acctype != AccType_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;

memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
ipa_64 = ZeroExtend(ipa.paddress.address<39:0>, 64);
// Output Address
oa = StageOA(ipa_64, 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)
integer tsize;
case sdftype of
    when SDFTYPE\_SmallPage      tsize = 12;
    when SDFTYPE\_LargePage     tsize = 16;
    when SDFTYPE\_Section       tsize = 20;
    when SDFTYPE\_Supersection  tsize = 24;

// Output Address
FullAddress oa;
oa.address = baseaddress.address<51: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, AccType acctype,
                                           boolean iswrite, boolean aligned,
                                           integer size)

regime = TranslationRegime(PSTATE.EL, acctype);
if !RegimeUsingAArch32(regime) then
    return AArch64.TranslateAddress(ZeroExtend(va, 64), acctype, iswrite,
                                   aligned, size);
result = AArch32.FullTranslate(va, acctype, iswrite, aligned);
if !IsFault(result) then
    result.fault = AArch32.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);

return result;

```

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_Page;
    elsif descriptor<1:0> == '11' then
        return DescriptorType_Table;
    elsif descriptor<1:0> == '01' && level != FINAL_LEVEL then
        return DescriptorType_Block;
    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> IN {'1x'} then
        return SDFTYPE_SmallPage;
    else
        return SDFTYPE_Invalid;
```

Library pseudocode for aarch32/translation/walk/AArch32.SIIASize

```
// AArch32.SIIASize()
// =====
// Retrieve the number of bits containing the input address for stage 1 translation

integer AArch32.SIIASize(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, SecurityState ss,
                                         SITTPParams walkparams, bits(32) va, boolean ispriv)

    FaultRecord fault = fault_in;
    bits(3) txsz;
    bits(64) ttbr;
    bit epd;
    if regime == Regime_EL2 then
        ttbr = HTTBR;
        txsz = walkparams.t0sz;
    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;
        elsif HaveAArch32EL(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;
baselsb = (iasize - (levels*stride + granulebits)) + 3;
baseaddress.paspace = if ss == SS_Secure then PAS_Secure else PAS_NonSecure;
baseaddress.address = ZeroExtend(ttbr<39:baselsb>;Zeros(baselsb));

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';

indexmsb = iasize - 1;
bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va);

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(ss) &&
    (if ELStateUsingAArch32(EL2, 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);

integer indexlsb;
DescriptorType desctype;
repeat
    fault.level = walkstate.level;
    indexlsb = (FINAL\_LEVEL - walkstate.level)*stride + granulebits;
    bits(40) index = ZeroExtend(va<indexmsb:indexlsb>:'000');

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

// 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(ss) then
    s2fslwalk = TRUE;
    s2acctype = AccType\_TTW;
    s2aligned = TRUE;
    s2write = FALSE;
    (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fslwalk,
                                                s2acctype, s2aligned, s2write, ispriv);

    // 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, fault);
else
    (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

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));
            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;
        indexmsb = indexlsb - 1;

        when DescriptorType\_Invalid
            fault.statuscode = Fault\_Translation;
            return (fault, ITWState UNKNOWN);

        when DescriptorType\_Page, DescriptorType\_Block
            walkstate.istable = FALSE;

until desctype IN {DescriptorType\_Page, DescriptorType\_Block};

// Check the output address is inside the supported range
if !IsZero(descriptor<47:40>) then
    fault.statuscode = Fault\_AddressSize;
    return (fault, ITWState UNKNOWN);

// Check the access flag
if descriptor<10> == '0' then
    fault.statuscode = Fault\_AccessFlag;
    return (fault, ITWState 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 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>;

walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));
if walkstate.baseaddress.paspace == PAS\_Secure && descriptor<5> == '1' then
    walkstate.baseaddress.paspace = PAS\_NonSecure;

memattr = descriptor<4:2>;
sh = descriptor<9:8>;
attr = MAIRAttr(UInt(memattr), walkparams.mair);
slaarch64 = FALSE;
walkstate.memattrs = S1DecodeMemAttrs(attr, sh, slaarch64);

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, SecurityState ss,
                                         bits(32) va, boolean ispriv)

    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;
    elsif HaveAArch32EL(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;
    n = UInt(ttbcr.N);
    bits(32) ttb;
    bits(1) pd;
    bits(2) irgn;
    bits(2) rgn;
    bits(1) s;
    bits(1) nos;
    if n == 0 || IsZero(va<31:(32-n)>) then
        ttb = ttbr0.TTB0:Zeros(7);
        pd = ttbcr.PD0;
        irgn = ttbr0.IRGN;
        rgn = ttbr0.RGN;
        s = ttbr0.S;
        nos = ttbr0.NOS;
    else
        n = 0; // TTBR1 translation always treats N as 0
        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 ss == SS_Secure then PAS_Secure else PAS_NonSecure;
    baseaddress.address = ZeroExtend(ttb<31:14-n>:Zeros(14-n));

    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       = 1;
walkstate.istable     = TRUE;

bits(4) domain;
bits(32) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = ZeroExtend(va);

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(ss) &&
    (if ELStateUsingAArch32(EL2, 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');
    else
        index = ZeroExtend(va<19:12>:'00');

    walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
    walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

    if regime == Regime_EL10 && AArch32.EL2Enabled(ss) then
        s2fslwalk = TRUE;
        s2acctype = AccType_TTW;
        s2aligned = TRUE;
        s2write = FALSE;
        (s2fault, s2walkaddress) = AArch32.S2Translate(fault, walkaddress, ss, s2fslwalk,
            s2acctype, s2aligned, s2write, ispriv);

        if s2fault.statuscode != Fault_None then
            return (s2fault, TTWState UNKNOWN);

        (fault, descriptor) = FetchDescriptor(ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, TTWState UNKNOWN);

    walkstate.sdftype = AArch32.DecodeDescriptorTypeSD(descriptor, walkstate.level);

    case walkstate.sdftype of
        when SDType_Invalid
            fault.domain = domain;

```

```

    fault.statuscode = Fault\_Translation;
    return (fault, ITWState UNKNOWN);

when SDType\_Table
    domain = descriptor<8:5>;
    ns     = descriptor<3>;
    pxn    = descriptor<2>;

    walkstate.baseaddress.address = ZeroExtend(descriptor<31:10>:Zeros(10));
    walkstate.level = 2;

when SDType\_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));
    walkstate.istable = FALSE;

when SDType\_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));
    walkstate.istable = FALSE;

when SDType\_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));
    walkstate.istable = FALSE;

when SDType\_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));
    walkstate.istable = FALSE;

until walkstate.sdftype != SDType\_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);
elseif 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 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

integer 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,
                                       AddressDescriptor ipa)
    FaultRecord fault = fault_in;

    if walkparams.sl0 IN {'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(VTTBR<47:40>) then
        fault.statuscode = Fault_AddressSize;
        fault.level      = 0;
        return (fault, TTWState UNKNOWN);

    FullAddress baseaddress;
    baselsb = (iasize - (levels*stride + granulebits)) + 3;
    baseaddress.paspace = PAS_NonSecure;
    baseaddress.address = ZeroExtend(VTTBR<39:baselsb>:Zeros(baselsb));

    TTWState walkstate;
    walkstate.baseaddress = baseaddress;
    walkstate.level       = startlevel;
    walkstate.istable     = TRUE;
    walkstate.memattrs    = WalkMemAttrs(walkparams.sh, walkparams.irgn,
                                         walkparams.orgn);

    indexmsb = iasize - 1;
    bits(64) descriptor;
    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 indexlsb;
    DescriptorType desctype;
    repeat
        fault.level = walkstate.level;

        indexlsb = (FINAL_LEVEL - walkstate.level)*stride + granulebits;
        bits(40) index = ZeroExtend(ipa.paddress.address<indexmsb:indexlsb>:'000');

        walkaddress.paddress.address = walkstate.baseaddress.address OR ZeroExtend(index);
        walkaddress.paddress.paspace = walkstate.baseaddress.paspace;

        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

        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));
        walkstate.level = walkstate.level + 1;
        indexmsb = indexlsb - 1;

    when DescriptorType_Invalid
        fault.statuscode = Fault_Translation;
        return (fault, TTWState UNKNOWN);

    when DescriptorType_Page, DescriptorType_Block
        walkstate.istable = FALSE;

until desctype IN {DescriptorType_Page, DescriptorType_Block};

// 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
walkstate.baseaddress.address = ZeroExtend(descriptor<39:indexlsb>:Zeros(indexlsb));

walkstate.permissions.s2ap = descriptor<7:6>;
walkstate.permissions.s2xn = descriptor<54>;
if HaveExtendedExecuteNeverExt() then
    walkstate.permissions.s2xnx = descriptor<53>;
else
    walkstate.permissions.s2xnx = '0';

memattr = descriptor<5:2>;
sh      = descriptor<9:8>;
walkstate.memattrs = S2DecodeMemAttrs(memattr, sh);
walkstate.contiguous = descriptor<52>;

return (fault, walkstate);

```

Library pseudocode for aarch32/translation/walk/AArch32.TranslationSizeSD

```

// AArch32.TranslationSizeSD()
// =====
// Determine the size of the translation

integer AArch32.TranslationSizeSD(SDFTType sdftype)
    integer tsize;
    case sdftype of
        when SDFTType_SmallPage      tsize = 12;
        when SDFTType_LargePage      tsize = 16;
        when SDFTType_Section        tsize = 20;
        when SDFTType_Supersection   tsize = 24;

    return tsize;

```

Library pseudocode for aarch32/translation/walk/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);

    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
    lo_iasize = AArch32.S1IASize(t0sz);
    // Upper range Input Address size
    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 HaveAArch32EL(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 HaveAArch32EL(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 HaveAArch32EL\(EL3\) 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 = SCR_EL3.SIF;

    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.ntlsm = if HaveTrapLoadStoreMultipleDeviceExt\(\) then sctlr.nTlsm 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 AArch32.HaveHPDExt\(\) then ttbcr.T2E AND ttbcr2.HPD0 else '0';
    else
        walkparams.sh = ttbcr.SH1;
        walkparams.irgn = ttbcr.IRGN1;
        walkparams.orgn = ttbcr.ORGNO;
        walkparams.hpd = if AArch32.HaveHPDExt\(\) 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 AArch32.HaveHPDExt() then HTCR.HPD else '0';
    walkparams.ee = HSCTLR.EE;
    walkparams.wxn = HSCTLR.WXN;
    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmd = HSCTLR.nTLSMD;
    else
        walkparams.ntlsmd = '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 = SCTL_S.EE;
    walkparams.wxn = SCTL_S.WXN;
    walkparams.uwxn = SCTL_S.UWXN;
    walkparams.ntlsmd = if HaveTrapLoadStoreMultipleDeviceExt() then SCTL_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 AArch32.HaveHPDExt() 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 AArch32.HaveHPDExt() then TTBCR_S.T2E AND TTBCR2_S.HPD1 else '0';

    return walkparams;
```

Library pseudocode for aarch64/debug/breakpoint/AArch64.BreakpointMatch

```
// AArch64.BreakpointMatch()
// =====
// Breakpoint matching in an AArch64 translation regime.

boolean AArch64.BreakpointMatch(integer n, bits(64) vaddress,
                                AccType acctype, integer size)
assert !ELUsingAArch32(S1TranslationRegime());
assert n < NumBreakpointsImplemented();

enabled = DBGBCR_EL1[n].E == '1';
ispriv = PSTATE.EL != EL0;
linked = DBGBCR_EL1[n].BT IN {'0x01'};
isbreakpnt = TRUE;
linked_to = FALSE;

state_match = AArch64.StateMatch(DBGBCR_EL1[n].SSC, DBGBCR_EL1[n].HMC, DBGBCR_EL1[n].PMC,
                                linked, DBGBCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
value_match = AArch64.BreakpointValueMatch(n, vaddress, linked_to);

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);
    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);

match = value_match && state_match && enabled;

return match;
```



```

// AArch64.BreakpointValueMatch()
// =====

boolean AArch64.BreakpointValueMatch(integer n_in, bits(64) 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;

// 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
    Constraint c;
    (c, n) = ConstrainUnpredictableInteger(0, NumBreakpointsImplemented() - 1, Unpredictable_BPNOTIMP);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return 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_EL1[n].E == '0' then return FALSE;

context_aware = (n >= (NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented()));

// If BT is set to a reserved type, behaves either as disabled or as a not-reserved type.
dbgtype = DBGBCR_EL1[n].BT;
Constraint c;

if ((dbgtype IN {'011x', '11xx'}) && !HaveVirtHostExt() && !HaveV82Debug()) || // Context matching
    dbgtype IN {'010x'} || // Reserved
    (!(dbgtype IN {'0x0x'}) && !context_aware) || // Context match
    (dbgtype IN {'1xxx'} && !HaveEL(EL2))) then // EL2 extension
    (c, dbgtype) = ConstrainUnpredictableBits(Unpredictable_RESBPTYPE);
    assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
    if c == Constraint_DISABLED then return FALSE;
    // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

// Determine what to compare against.
match_addr = (dbgtype IN {'0x0x'});
match_vmid = (dbgtype IN {'10xx'});
match_cid = (dbgtype IN {'001x'});
match_cid1 = (dbgtype IN {'101x', 'x11x'});
match_cid2 = (dbgtype IN {'11xx'});
linked = (dbgtype IN {'xxx1'});

// If this is a call from StateMatch, return FALSE if the breakpoint is not programmed for a
// VMID and/or context ID match, or if not context-aware. The above assertions mean that the
// code can just test for match_addr == TRUE to confirm all these things.
if linked_to && (!linked || match_addr) then return FALSE;

// If called from BreakpointMatch return FALSE for Linked context ID and/or VMID matches.
if !linked_to && linked && !match_addr then return FALSE;

// Do the comparison.
boolean BVR_match;
if match_addr then
    boolean byte_select_match;
    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
// If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
// of DBGBVR<n>_EL1.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.
integer top = AArch64.VAMax();
if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
    if ConstrainUnpredictableBool(Unpredictable_DBGxVR_RESS) then
        top = 63;
BVR_match = (vaddress<top:2> == DBGBVR_EL1[n]<top:2>) && byte_select_match;

elseif match_cid then
    if IsInHost() then
        BVR_match = (CONTEXTIDR_EL2<31:0> == DBGBVR_EL1[n]<31:0>);
    else
        BVR_match = (PSTATE.EL IN {EL0, EL1} && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
elseif match_cid1 then
    BVR_match = (PSTATE.EL IN {EL0, EL1} && !IsInHost() && CONTEXTIDR_EL1<31:0> == DBGBVR_EL1[n]<31:0>);
boolean BXVR_match;
if match_vmid then
    bits(16) vmid;
    bits(16) bvr_vmid;
    if !Have16bitVMID() || VTCR_EL2.VS == '0' then
        vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
        bvr_vmid = ZeroExtend(DBGBVR_EL1[n]<39:32>, 16);
    else
        vmid = VTTBR_EL2.VMID;
        bvr_vmid = DBGBVR_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 && (HaveVirtHostExt() || HaveV82Debug()) &&
        EL2Enabled() &&
        DBGBVR_EL1[n]<63:32> == CONTEXTIDR_EL2<31:0>);

bvr_match_valid = (match_addr || match_cid || match_cid1);
bxvr_match_valid = (match_vmid || match_cid2);

match = (!bxvr_match_valid || BXVR_match) && (!bvr_match_valid || BVR_match);

return match;

```



```

// AArch64.StateMatch()
// =====
// Determine whether a breakpoint or watchpoint is enabled in the current mode and state.

boolean AArch64.StateMatch(bits(2) SSC_in, bit HMC_in,
                           bits(2) PxC_in, boolean linked_in, bits(4) LBN,
                           boolean isbreakpt, AccType acctype, boolean ispriv)

// "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.
// "LBN" is the linked breakpoint number from the DBGBCR[n] or DBGWCR[n] register.
// "isbreakpt" is TRUE for breakpoints, FALSE for watchpoints.
// "ispriv" is valid for watchpoints, and selects between privileged and unprivileged accesses.
bits(2) SSC = SSC_in;
bit HMC = HMC_in;
bits(2) PxC = PxC_in;
boolean linked = linked_in;

// If parameters are set to a reserved type, behaves as either disabled or a defined type
Constraint c;
(c, SSC, HMC, PxC) = CheckValidStateMatch(SSC, HMC, PxC, isbreakpt);
if c == Constraint\_DISABLED then return FALSE;
// Otherwise the HMC,SSC,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;
if HaveNV2Ext() && acctype == AccType\_NV2REGISTER && !isbreakpt then
    priv_match = EL2_match;
elseif !ispriv && !isbreakpt then
    priv_match = EL0_match;
else
    case PSTATE.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;

boolean security_state_match;
ss = CurrentSecurityState();
case SSC of
    when '00' security_state_match = TRUE; // Both
    when '01' security_state_match = ss == SS\_NonSecure; // Non-secure only
    when '10' security_state_match = ss == SS\_Secure; // Secure only
    when '11' security_state_match = (HMC == '1' || ss == SS\_Secure); // HMC=1 -> Both, 0 -> Secure

integer lbn;
if linked then
    // "LBN" must be an enabled context-aware breakpoint unit. If it is not context-aware then
    // it is CONSTRAINED UNPREDICTABLE whether this gives no match, or LBN is mapped to some
    // UNKNOWN breakpoint that is context-aware.
    lbn = UInt(LBN);
    first_ctx_cmp = NumBreakpointsImplemented() - NumContextAwareBreakpointsImplemented();
    last_ctx_cmp = NumBreakpointsImplemented() - 1;
    if (lbn < first_ctx_cmp || lbn > last_ctx_cmp) then
        (c, lbn) = ConstrainUnpredictableInteger(first_ctx_cmp, last_ctx_cmp, Unpredictable\_BPNOTCTX);
        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

boolean linked_match;
if linked then
    vaddress = bits(64) UNKNOWN;
    linked_to = TRUE;
    linked_match = AArch64.BreakpointValueMatch(lbn, vaddress, linked_to);

```

```
return priv_match && security_state_match && (!linked || linked_match);
```

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 == EL0 && ELUsingAArch32\(EL1\) then
            enabled = enabled || SDER32\_EL3.SUIDEN == '1';
    else
        enabled = TRUE;

    if from_el == target then
        enabled = enabled && MDCR\_EL1.KDE == '1' && mask == '0';
    else
        enabled = enabled && UInt\(target\) > UInt\(from\_el\);

    return enabled;
```

Library pseudocode for aarch64/debug/pmu/AArch64.CheckForPMUOverflow

```
// AArch64.CheckForPMUOverflow()
// =====
// Signal Performance Monitors overflow IRQ and CTI overflow events

AArch64.CheckForPMUOverflow()
    boolean pmuirq;
    bit E;
    pmuirq = PMCR\_EL0.E == '1' && PMINTENSET\_EL1.C == '1' && PMOVSSET\_EL0.C == '1';
    integer counters = GetNumEventCounters\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            E = if AArch64.PMUCounterIsHyp(idx) then MDCR\_EL2.HPME else PMCR\_EL0.E;
            if E == '1' && PMINTENSET\_EL1<idx> == '1' && PMOVSSET\_EL0<idx> == '1' then pmuirq = TRUE;

    SetInterruptRequestLevel(InterruptID\_PMIU\_IRQ, if pmuirq then HIGH else LOW);

    CTI\_SetEventLevel(CrossTriggerIn\_PMUOverflow, if pmuirq then HIGH else 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 PMOVSCLR\_EL0.)
```

Library pseudocode for aarch64/debug/pmu/AArch64.ClearEventCounters

```
// AArch64.ClearEventCounters()
// =====
// Zero all the event counters.

AArch64.ClearEventCounters()
    integer counters = AArch64.GetNumEventCountersAccessible\(\);
    if counters != 0 then
        for idx = 0 to counters - 1
            PMEVCNTR_EL0[idx] = Zeros\(\);
```



```

// AArch64.CountPMUEvents()
// =====
// Return TRUE if counter "idx" should count its event.
// For the cycle counter, idx == CYCLE_COUNTER_ID.

boolean AArch64.CountPMUEvents(integer idx)
    assert idx == CYCLE\_COUNTER\_ID || idx < GetNumEventCounters\(\);
    boolean debug;
    boolean enabled;
    boolean prohibited;
    boolean filtered;
    boolean frozen;
    boolean resvd_for_el2;
    bit E;
    bit spme;
    bits(32) ovflws;
    // Event counting is disabled in Debug state
    debug = Halted\(\);

    // Software can reserve some counters for EL2
    resvd_for_el2 = AArch64.PMUCounterIsHyp(idx);
    ss = CurrentSecurityState();

    // Main enable controls
    if idx == CYCLE\_COUNTER\_ID then
        enabled = PMCR_EL0.E == '1' && PMCNTENSET_EL0.C == '1';
    else
        E = if resvd_for_el2 then MDCR_EL2.HPME else PMCR_EL0.E;
        enabled = E == '1' && PMCNTENSET_EL0<idx> == '1';

    // Event counting is allowed unless it is prohibited by any rule below
    prohibited = FALSE;

    // Event counting in Secure state is prohibited if all of:
    // * EL3 is implemented
    // * MDCR_EL3.SPME == 0, and either:
    //   - FEAT_PMUv3p7 is not implemented
    //   - MDCR_EL3.MPMX == 0
    if HaveEL(EL3) && ss == SS\_Secure then
        if HavePMUv3p7() then
            prohibited = MDCR_EL3.<SPME,MPMX> == '00';
        else
            prohibited = MDCR_EL3.SPME == '0';

    // Event counting at EL3 is prohibited if all of:
    // * FEAT_PMUv3p7 is implemented
    // * One of the following is true:
    //   - MDCR_EL3.SPME == 0
    //   - PMNx is not reserved for EL2
    // * MDCR_EL3.MPMX == 1
    if !prohibited && PSTATE.EL == EL3 && HavePMUv3p7() then
        prohibited = MDCR_EL3.MPMX == '1' && (MDCR_EL3.SPME == '0' || !resvd_for_el2);

    // Event counting at EL2 is prohibited if all of:
    // * The HPMD Extension is implemented
    // * PMNx is not reserved for EL2
    // * MDCR_EL2.HPMD == 1
    if !prohibited && PSTATE.EL == EL2 && HaveHPMDExt() && !resvd_for_el2 then
        prohibited = MDCR_EL2.HPMD == '1';

    // The IMPLEMENTATION DEFINED authentication interface might override software
    if prohibited && !HaveNoSecurePMUDisableOverride() then
        prohibited = !ExternalSecureNoninvasiveDebugEnabled();

    // Event counting might be frozen
    frozen = FALSE;

    // If FEAT_PMUv3p7 is implemented, event counting can be frozen
    if HavePMUv3p7() then
        ovflws = ZeroExtend(PMOVSSET_EL0<GetNumEventCounters()-1:0>);

```

```

bit FZ;
if resvd_for_el2 then
    FZ = MDCR_EL2.HPMFZ0;
    ovflws<UInt(MDCR_EL2.HPMN)-1:0> = Zeros();
else
    FZ = PMCR_EL0.FZ0;
    if HaveEL(EL2) && UInt(MDCR_EL2.HPMN) < GetNumEventCounters() then
        ovflws<GetNumEventCounters()-1:UInt(MDCR_EL2.HPMN)> = Zeros();
    frozen = (FZ == '1') && !IsZero(ovflws);

// PMCR_EL0.DP disables the cycle counter when event counting is prohibited
if (prohibited || frozen) && idx == CYCLE_COUNTER_ID then
    enabled = enabled && (PMCR_EL0.DP == '0');
    // Otherwise whether event counting is prohibited 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 HavePMUv3p5() && idx == CYCLE_COUNTER_ID then
    if (HaveEL(EL3) && ss == SS_Secure &&
        MDCR_EL3.SCCD == '1') then
        prohibited = TRUE;
    if PSTATE.EL == EL2 && MDCR_EL2.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 HavePMUv3p7() && idx == CYCLE_COUNTER_ID then
    if PSTATE.EL == EL3 && MDCR_EL3.MCCD == '1' then
        prohibited = TRUE;

// Event counting can be filtered by the {P, U, NSK, NSU, NSH, M, SH} bits
filter = if idx == CYCLE_COUNTER_ID then PMCCFILTR_EL0<31:0> else PMEVTYPER_EL0[idx]<31:0>;

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) then filter<26> else '0';
SH = if HaveEL(EL3) && HaveSecureEL2Ext() then filter<24> else '0';

ss = CurrentSecurityState();
case PSTATE.EL of
    when EL0 filtered = if ss == SS_Secure then U == '1' else U != NSU;
    when EL1 filtered = if ss == SS_Secure then P == '1' else P != NSK;
    when EL2 filtered = if ss == SS_Secure then NSH == SH else NSH == '0';
    when EL3 filtered = M != P;

return !debug && enabled && !prohibited && !filtered && !frozen;

```

Library pseudocode for aarch64/debug/pmu/AArch64.GetNumEventCountersAccessible

```
// AArch64.GetNumEventCountersAccessible()
// =====
// Return the number of event counters that can be accessed at the current Exception level.

integer AArch64.GetNumEventCountersAccessible()
    integer n;
    integer total_counters = GetNumEventCounters\(\);
    // Software can reserve some counters for EL2
    if PSTATE.EL IN {EL1, EL0} && EL2Enabled\(\) then
        n = UInt(MDCR_EL2.HPMN);
        if n > total_counters || (!HaveFeatHPMN0\(\) && n == 0) then
            (-, n) = ConstrainUnpredictableInteger(0, total_counters,
                Unpredictable\_PMUEVENTCOUNTER);
    else
        n = total_counters;

    return n;
```

Library pseudocode for aarch64/debug/pmu/AArch64.IncrementEventCounter

```
// AArch64.IncrementEventCounter()
// =====
// Increment the specified event counter by the specified amount.

AArch64.IncrementEventCounter(integer idx, integer increment)
    integer old_value;
    integer new_value;
    integer ovflw;
    bit lp;
    old_value = UInt(PMEVCNTR_EL0[idx]);
    new_value = old_value + PMUCountValue(idx, increment);

    if HavePMUv3p5\(\) then
        PMEVCNTR_EL0[idx] = new_value<63:0>;
        lp = if AArch64.PMUCounterIsHyp(idx) then MDCR_EL2.HLP else PMCR_EL0.LP;
        ovflw = if lp == '1' then 64 else 32;
    else
        PMEVCNTR_EL0[idx] = ZeroExtend(new_value<31:0>);
        ovflw = 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 < GetNumEventCounters\(\) && (!HavePMUv3p5\(\) || lp == '0') then
            PMUEvent(PMU_EVENT_CHAIN, 1, idx + 1);
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUCounterIsHyp

```
// AArch64.PMUCounterIsHyp
// =====
// Returns TRUE if a counter is reserved for use by EL2, FALSE otherwise.

boolean AArch64.PMUCounterIsHyp(integer n)
    boolean resvd_for_el2;
    // Software can reserve some event counters for EL2
    if n != CYCLE\_COUNTER\_ID && HaveEL(EL2) then
        resvd_for_el2 = n >= UInt(MDCR_EL2.HPMN);
        if UInt(MDCR_EL2.HPMN) > GetNumEventCounters\(\) || (!HaveFeatHPMN0\(\) && IsZero(MDCR_EL2.HPMN)) then
            resvd_for_el2 = boolean UNKNOWN;
    else
        resvd_for_el2 = FALSE;

    return resvd_for_el2;
```

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 !HavePMUv3() then
        return;

    PMUEvent(PMU_EVENT_CPU_CYCLES);

    integer counters = GetNumEventCounters();
    if counters != 0 then
        for idx = 0 to counters - 1
            if AArch64.CountPMUEvents(idx) then
                accumulated = PMUEventAccumulator[idx];
                AArch64.IncrementEventCounter(idx, accumulated);
                PMUEventAccumulator[idx] = 0;

    integer old_value;
    integer new_value;
    integer ovflw;
    if (AArch64.CountPMUEvents(CYCLE_COUNTER_ID) &&
        (!HaveAArch32() || PMCR_EL0.LC == '1' || PMCR_EL0.D == '0' || HasElapsed64Cycles())) then
        old_value = UInt(PMCCNTR_EL0);
        new_value = old_value + 1;
        PMCCNTR_EL0 = new_value<63:0>;

    if HaveAArch32() then
        ovflw = if PMCR_EL0.LC == '1' then 64 else 32;
    else
        ovflw = 64;

    if old_value<64:ovflw> != new_value<64:ovflw> then
        PMOVSSET_EL0.C = '1';
        PMOVSCLR_EL0.C = '1';

    AArch64.CheckForPMUOverflow();
```

Library pseudocode for aarch64/debug/pmu/AArch64.PMUSwIncrement

```
// AArch64.PMUSwIncrement()
// =====
// Generate PMU Events on a write to PMSWINC_EL0.

AArch64.PMUSwIncrement(bits(32) sw_incr)
    integer counters = AArch64.GetNumEventCountersAccessible();
    if counters != 0 then
        for idx = 0 to counters - 1
            if sw_incr<idx> == '1' then
                PMUEvent(PMU_EVENT_SW_INCR, 1, idx);
```

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 !HaveECVExt() 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);
  if EL2Enabled() then
    bits(2) PCT_el2;
    if !HaveECVExt() 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);
  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 HaveECVExt(); // 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 HaveECVExt(); // FEAT_ECV must be implemented
      return TimeStamp_OffsetPhysical;
    otherwise Unreachable();
```

Library pseudocode for aarch64/debug/statisticalprofiling/OpType

```
enumeration OpType {
  OpType_Load, // Any memory-read operation other than atomics, compare-and-swap, and swap
  OpType_Store, // Any memory-write operation, including atomics without return
  OpType_LoadAtomic, // Atomics with return, compare-and-swap and swap
  OpType_Branch, // Software write to the PC
  OpType_Other // Any other class of operation
};
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferEnabled

```
// ProfilingBufferEnabled()
// =====

boolean ProfilingBufferEnabled()
  if !HaveStatisticalProfiling() then return FALSE;
  (owning_ss, owning_el) = ProfilingBufferOwner();
  state_match = ((owning_ss == SS_Secure    && SCR_EL3.NS == '0') ||
                 (owning_ss == SS_NonSecure && SCR_EL3.NS == '1'));
  return (!ELUsingArch32(owning_el) && state_match &&
          PMBLIMITR_EL1.E == '1' && PMBSR_EL1.S == '0');
```

Library pseudocode for aarch64/debug/statisticalprofiling/ProfilingBufferOwner

```
// ProfilingBufferOwner()
// =====

(SecurityState, bits(2)) ProfilingBufferOwner()
  SecurityState owning_ss;

  if HaveEL(EL3) then
    owning_ss = if MDCR_EL3.NSPB<1> == '0' then SS_Secure else SS_NonSecure;
  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

```
// 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/StatisticalProfilingEnabled

```
// StatisticalProfilingEnabled()
// =====

boolean StatisticalProfilingEnabled()
  if !HaveStatisticalProfiling() || UsingArch32() || !ProfilingBufferEnabled() then
    return FALSE;

  tge_set = EL2Enabled() && HCR_EL2.TGE == '1';
  (owning_ss, owning_el) = ProfilingBufferOwner();
  if (UInt(owning_el) < UInt(PSTATE.EL) || (tge_set && owning_el == EL1) ||
      owning_ss != CurrentSecurityState()) then
    return FALSE;
  bit spe_bit;
  case PSTATE.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

```
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
```

Library pseudocode for aarch64/debug/takeexceptiondbg/AArch64.TakeExceptionInDebugState

```
// 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);
    ExceptionRecord exception = exception_in;
    boolean sync_errors;
    boolean iesb_req;
    if HaveIESB() then
        sync_errors = SCTLR[target_el].IESB == '1';
        if HaveDoubleFaultExt() then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
        // SCTLR[].IESB and/or SCR_EL3.NMEA (if applicable) might be ignored in Debug state.
        if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
            sync_errors = FALSE;
    else
        sync_errors = FALSE;

    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();

    AArch64.ReportException(exception, target_el);

    PSTATE.EL = target_el;
    PSTATE.nRW = '0';
    PSTATE.SP = '1';

    SPSR[] = bits(64) UNKNOWN;
    ELR[] = 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 (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
        SCTLR[].SPAN == '0') then
        PSTATE.PAN = '1';
    if HaveUAQExt() then PSTATE.UA0 = '0';
    if HaveBTIExt() then PSTATE.BTYPE = '00';
    if HaveSSBSExt() then PSTATE.SSBS = bit UNKNOWN;
    if HaveMTEEExt() then PSTATE.TCO = '1';

    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, AccType acctype, bits(64) vaddress)

integer top = AArch64.VAMax();
bottom = if DBGWVR_EL1[n]<2> == '1' then 2 else 3;           // Word or doubleword
byte_select_match = (DBGWCR_EL1[n].BAS<UInt(vaddress<bottom-1:0>)> != '0');
mask = UInt(DBGWCR_EL1[n].MASK);

// If DBGWCR_EL1[n].MASK is non-zero 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

boolean WVR_match;
if mask > bottom then
    // If the DBGxVR<n>_EL1.RESS field bits are not a sign extension of the MSB
    // of DBGBVR<n>_EL1.VA, it is UNPREDICTABLE whether they appear to be
    // included in the match.
    if !IsOnes(DBGBVR_EL1[n]<63:top>) && !IsZero(DBGBVR_EL1[n]<63:top>) then
        if ConstrainUnpredictableBool(Unpredictable\_DBGxVR\_RESS) then
            top = 63;
    WVR_match = (vaddress<top:mask> == DBGWVR_EL1[n]<top:mask>);
    // If masked bits of DBGWVR_EL1[n] are not zero, the behavior is CONSTRAINED UNPREDICTABLE.
    if WVR_match && !IsZero(DBGWVR_EL1[n]<mask-1:bottom>) then
        WVR_match = ConstrainUnpredictableBool(Unpredictable\_WPMASKEDBITS);
else
    WVR_match = vaddress<top:bottom> == DBGWVR_EL1[n]<top:bottom>;

return WVR_match && byte_select_match;
```

Library pseudocode for aarch64/debug/watchpoint/AArch64.WatchpointMatch

```
// AArch64.WatchpointMatch()
// =====
// Watchpoint matching in an AArch64 translation regime.

boolean AArch64.WatchpointMatch(integer n, bits(64) vaddress, integer size, boolean ispriv,
                                AccType acctype, boolean iswrite)
    assert !ELUsingAArch32(S1TranslationRegime());
    assert n < NumWatchpointsImplemented();

    // "ispriv" is:
    // * FALSE for all loads, stores, and atomic operations executed at EL0.
    // * FALSE if the access is unprivileged.
    // * TRUE for all other loads, stores, and atomic operations.

    enabled = DBGWCR_EL1[n].E == '1';
    linked = DBGWCR_EL1[n].WT == '1';
    isbreakpnt = FALSE;

    state_match = AArch64.StateMatch(DBGWCR_EL1[n].SSC, DBGWCR_EL1[n].HMC, DBGWCR_EL1[n].PAC,
                                     linked, DBGWCR_EL1[n].LBN, isbreakpnt, acctype, ispriv);
    ls_match = FALSE;
    if IsAtomicRW(acctype) then
        ls_match = (DBGWCR_EL1[n].LSC != '00');
    elsif iswrite then
        ls_match = (DBGWCR_EL1[n].LSC<1> != '0');
    else
        ls_match = (DBGWCR_EL1[n].LSC<0> != '0');

    value_match = FALSE;
    for byte = 0 to size - 1
        value_match = value_match || AArch64.WatchpointByteMatch(n, acctype, vaddress + byte);

    return value_match && state_match && ls_match && enabled;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.Abort

```
// AArch64.Abort()
// =====
// Abort and Debug exception handling in an AArch64 translation regime.

AArch64.Abort(bits(64) vaddress, FaultRecord fault)

    if IsDebugException(fault) then
        if fault.acctype == AccType_IFETCH then
            if UsingAArch32() && fault.debugmoe == DebugException_VectorCatch then
                AArch64.VectorCatchException(fault);
            else
                AArch64.BreakpointException(fault);
        else
            AArch64.WatchpointException(vaddress, fault);
    elsif fault.acctype == AccType_IFETCH then
        AArch64.InstructionAbort(vaddress, fault);
    else
        AArch64.DataAbort(vaddress, 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(64) vaddress)
    exception = ExceptionSyndrome(exceptype);

    d_side = exceptype IN {Exception_DataAbort, Exception_NV2DataAbort, Exception_Watchpoint, Exception_M
    (exception.syndrome, exception.syndrome2) = AArch64.FaultSyndrome(d_side, fault);
    exception.vaddress = ZeroExtend(vaddress);
    if IPAValid(fault) then
        exception.ipavalid = TRUE;
        exception.NS = if fault.ipaddress.paspace == PAS_NonSecure then '1' else '0';
        exception.ipaddress = fault.ipaddress.address;
    else
        exception.ipavalid = FALSE;

    return exception;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.CheckPCAlignment

```
// AArch64.CheckPCAlignment()
// =====

AArch64.CheckPCAlignment()

    bits(64) pc = ThisInstrAddr();
    if pc<1:0> != '00' then
        AArch64.PCAlignmentFault();
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.DataAbort

```
// AArch64.DataAbort()
// =====

AArch64.DataAbort(bits(64) vaddress, FaultRecord fault)
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
        (HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER) ||
        IsSecondStage(fault)));

    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;
    ExceptionRecord exception;
    if HaveNV2Ext() && fault.acctype == AccType_NV2REGISTER then
        exception = AArch64.AbortSyndrome(Exception_NV2DataAbort, fault, vaddress);
    else
        exception = AArch64.AbortSyndrome(Exception_DataAbort, fault, vaddress);
    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.EffectiveTCF

```
// AArch64.EffectiveTCF()
// =====
// Returns the TCF field applied to tag check faults in the given Exception level.

bits(2) AArch64.EffectiveTCF(AccType acctype)
    bits(2) tcf, el;
    el = S1TranslationRegime();

    if el == EL3 then
        tcf = SCTLR_EL3.TCF;
    elsif el == EL2 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL2.TCF0;
        else
            tcf = SCTLR_EL2.TCF;
    elsif el == EL1 then
        if AArch64.AccessUsesEL(acctype) == EL0 then
            tcf = SCTLR_EL1.TCF0;
        else
            tcf = SCTLR_EL1.TCF;

    if tcf == '11' then //reserved value
        if !HaveMTE3Ext() then
            (-,tcf) = ConstrainUnpredictableBits(Unpredictable_RESTCF);

    return tcf;
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.InstructionAbort

```
// AArch64.InstructionAbort()
// =====

AArch64.InstructionAbort(bits(64) vaddress, FaultRecord fault)
    // External aborts on instruction fetch must be taken synchronously
    if HaveDoubleFaultExt() then assert fault.statuscode != Fault\_AsyncExternal;
    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1' && IsExternalAbort(fault);
    route_to_el2 = (EL2Enabled() && PSTATE.EL IN {EL0, EL1} &&
        (HCR_EL2.TGE == '1' ||
        (HaveRASExt() && HCR_EL2.TEA == '1' && IsExternalAbort(fault)) ||
        IsSecondStage(fault)));

    ExceptionRecord exception;
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset;
    if (HaveDoubleFaultExt() && (PSTATE.EL == EL3 || route_to_el3) &&
        IsExternalAbort(fault) && SCR_EL3.EASE == '1') then
        vect_offset = 0x180;
    else
        vect_offset = 0x0;

    exception = AArch64.AbortSyndrome(Exception\_InstructionAbort, fault, vaddress);

    bits(2) target_el = EL1;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    AArch64.TakeException(target_el, exception, 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()

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_PCAlignment\);
    exception.vaddress = ThisInstrAddr\(\);

    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, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.RaiseTagCheckFault

```
// AArch64.RaiseTagCheckFault()
// =====
// Raise a tag check fault exception.

AArch64.RaiseTagCheckFault(bits(64) va, boolean write)
    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    integer vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_DataAbort\);
    exception.syndrome<5:0> = '010001';
    if write then
        exception.syndrome<6> = '1';
    exception.vaddress = bits(4) UNKNOWN : va<59:0>;

    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, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.ReportTagCheckFault

```
// AArch64.ReportTagCheckFault()
// =====
// Records a tag check fault exception into the appropriate TCFR_ELx.

AArch64.ReportTagCheckFault(bits(2) el, bit ttbr)
    if el == EL3 then
        assert ttbr == '0';
        TFSR_EL3.TF0 = '1';
    elseif el == EL2 then
        if ttbr == '0' then
            TFSR_EL2.TF0 = '1';
        else
            TFSR_EL2.TF1 = '1';
    elseif el == EL1 then
        if ttbr == '0' then
            TFSR_EL1.TF0 = '1';
        else
            TFSR_EL1.TF1 = '1';
    elseif el == EL0 then
        if ttbr == '0' then
            TFSRE0_EL1.TF0 = '1';
        else
            TFSRE0_EL1.TF1 = '1';
```

Library pseudocode for aarch64/exceptions/aborts/AArch64.SPAlignmentFault

```
// AArch64.SPAlignmentFault()
// =====
// Called on an unaligned stack pointer in AArch64 state.

AArch64.SPAlignmentFault()

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_SPAlignment\);

    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, exception, 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, AccType acctype, boolean iswrite)
    bits(2) tcf, el;
    el = AArch64.AccessUsesEL(acctype);
    tcf = AArch64.EffectiveTCF(acctype);
    case tcf of
        when '00' // Tag Check Faults have no effect on the PE
            return;
        when '01' // Tag Check Faults cause a synchronous exception
            AArch64.RaiseTagCheckFault(vaddress, iswrite);
        when '10' // Tag Check Faults are asynchronously accumulated
            AArch64.ReportTagCheckFault(el, vaddress<55>);
        when '11' // Tag Check Faults cause a synchronous exception on reads or on
                // a read/write access, and are asynchronously accumulated on writes
            // Check for access performing both a read and a write.
            if !iswrite || IsAtomicRW(acctype) then
                AArch64.RaiseTagCheckFault(vaddress, iswrite);
            else
                AArch64.ReportTagCheckFault(PSTATE.EL, vaddress<55>);
```

Library pseudocode for aarch64/exceptions/aborts/BranchTargetException

```
// BranchTargetException()
// =====
// Raise branch target exception.

AArch64.BranchTargetException(bits(52) vaddress)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_BranchTarget);
    exception.syndrome<1:0> = PSTATE.BTYPE;
    exception.syndrome<24:2> = Zeros(); // 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, exception, 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'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;
    exception = ExceptionSyndrome(Exception_FIQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, 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'));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    if route_to_el3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    elsif PSTATE.EL == EL2 || route_to_el2 then
        assert PSTATE.EL != EL3;
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        assert PSTATE.EL IN {EL0, EL1};
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakePhysicalSErrorException

```
// AArch64.TakePhysicalSErrorException()
// =====

AArch64.TakePhysicalSErrorException(bits(25) syndrome)

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled() &&
                    (HCR_EL2.TGE == '1' || (!IsInHost() && HCR_EL2.AMO == '1')));
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;

    bits(2) target_el;
    if PSTATE.EL == EL3 || route_to_el3 then
        target_el = EL3;
    elsif PSTATE.EL == EL2 || route_to_el2 then
        target_el = EL2;
    else
        target_el = EL1;

    if IsSErrorEdgeTriggered(target_el, syndrome) then
        ClearPendingPhysicalSError();

    exception = ExceptionSyndrome(Exception_SError);
    exception.syndrome = syndrome;
    AArch64.TakeException(target_el, exception, 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

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x100;

    exception = ExceptionSyndrome(Exception_FIQ);

    AArch64.TakeException(EL1, exception, 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

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x80;

    exception = ExceptionSyndrome(Exception_IRQ);

    AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/async/AArch64.TakeVirtualSErrorException

```
// AArch64.TakeVirtualSErrorException()
// =====

AArch64.TakeVirtualSErrorException(bits(25) syndrome)

    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

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x180;
    exception = ExceptionSyndrome(Exception_SError);

    if HaveRASExt() then
        exception.syndrome<24> = VESR_EL2.IDS;
        exception.syndrome<23:0> = VESR_EL2.ISS;
    else
        impdef_syndrome = syndrome<24> == '1';
        if impdef_syndrome then exception.syndrome = syndrome;

    ClearPendingVirtualSError();
    AArch64.TakeException(EL1, exception, 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'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_Breakpoint, fault, vaddress);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, 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'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareBreakpoint);
    exception.syndrome<15:0> = immediate;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, 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'));

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_SoftwareStep);
    if SoftwareStep_DidNotStep() then
        exception.syndrome<24> = '0';
    else
        exception.syndrome<24> = '1';
        exception.syndrome<6> = if SoftwareStep_SteppedEX() then '1' else '0';
    exception.syndrome<5:0> = '100010'; // IFSC = Debug Exception

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, 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');

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    vaddress = bits(64) UNKNOWN;
    exception = AArch64.AbortSyndrome(Exception_VectorCatch, fault, vaddress);

    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/debug/AArch64.WatchpointException

```
// AArch64.WatchpointException()
// =====

AArch64.WatchpointException(bits(64) vaddress, FaultRecord fault)
    assert PSTATE.EL != EL3;

    route_to_el2 = (PSTATE.EL IN {EL0, EL1} && EL2Enabled\(\) &&
        (HCR_EL2.TGE == '1' || MDCR_EL2.TDE == '1'));

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    ExceptionRecord exception;
    if HaveNV2Ext\(\) && fault.acctype == AccType\_NV2REGISTER then
        exception = AArch64.AbortSyndrome\(Exception\_NV2Watchpoint, fault, vaddress\);
    else
        exception = AArch64.AbortSyndrome\(Exception\_Watchpoint, fault, vaddress\);

    if PSTATE.EL == EL2 || route_to_el2 then
        AArch64.TakeException\(EL2, exception, preferred\_exception\_return, vect\_offset\);
    else
        AArch64.TakeException\(EL1, exception, preferred\_exception\_return, vect\_offset\);
```

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_WFxTrap                ec = 0x01;
        when Exception_CP15RITrap             ec = 0x03; assert from_32;
        when Exception_CP15RRTTrap           ec = 0x04; assert from_32;
        when Exception_CP14RTTrap            ec = 0x05; assert from_32;
        when Exception_CP14DTTrap            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_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_ERetTrap              ec = 0x1A; assert !from_32;
        when Exception_PACFail                ec = 0x1C; 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_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;
        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 exception, bits(2) target_el)

    Exception exceptype = exception.exceptype;

    (ec,il) = AArch64.ExceptionClass(exceptype, target_el);
    iss = exception.syndrome;
    iss2 = exception.syndrome2;

    // 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[target_el] = (Zeros(27) : // <63:37>
                    iss2      : // <36:32>
                    ec<5:0>   : // <31:26>
                    il        : // <25>
                    iss);     // <24:0>

    if exceptype IN {
        Exception_InstructionAbort,
        Exception_PCAlignment,
        Exception_DataAbort,
        Exception_NV2DataAbort,
        Exception_NV2Watchpoint,
        Exception_Watchpoint
    } then
        FAR[target_el] = exception.vaddress;
    else
        FAR[target_el] = bits(64) UNKNOWN;

    if exception.ipavalid then
        HPFAR_EL2<43:4> = exception.ipaddress<51:12>;
        if IsSecureEL2Enabled() && CurrentSecurityState() == SS_Secure then
            HPFAR_EL2.NS = exception.NS;
        else
            HPFAR_EL2.NS = '0';
    elsif target_el == EL2 then
        HPFAR_EL2<43:4> = bits(40) UNKNOWN;

    return;
```

Library pseudocode for aarch64/exceptions/exceptions/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
    PSTATE.IL = '0';          // Clear Illegal Execution state bit

    // All registers, bits and fields not reset by the above pseudocode or by the BranchTo() call
    // below are UNKNOWN bitstrings after reset. In particular, the return information registers
    // ELR_ELx and SPSR_ELx have UNKNOWN values, so that it
    // is impossible to return from a reset in an architecturally defined way.
    AArch64.ResetGeneralRegisters();
    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
    assert IsZero(rv<63:AArch64.PAMax()>) && IsZero(rv<1:0>);

    boolean branch_conditional = FALSE;
    BranchTo(rv, BranchType\_RESET, branch_conditional);
```

Library pseudocode for aarch64/exceptions/ieeefp/AArch64.FPTrappedException

```
// AArch64.FPTrappedException()
// =====

AArch64.FPTrappedException(boolean is_ase, bits(8) accumulated_exceptions)
    exception = ExceptionSyndrome\(Exception\_FPTrappedException\);
    if is_ase then
        if boolean IMPLEMENTATION_DEFINED "vector instructions set TFV to 1" then
            exception.syndrome<23> = '1'; // TFV
        else
            exception.syndrome<23> = '0'; // TFV
    else
        exception.syndrome<23> = '1'; // TFV
    exception.syndrome<10:8> = bits(3) UNKNOWN; // VECITR
    if exception.syndrome<23> == '1' then
        exception.syndrome<7,4:0> = accumulated_exceptions<7,4:0>; // IDF,IXF,UFF,OFF,DZF,IOF
    else
        exception.syndrome<7,4:0> = bits(6) UNKNOWN;

    route_to_el2 = EL2Enabled\(\) && HCR_EL2.TGE == '1';

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, 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();
    bits(64) preferred_exception_return = NextInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_HypervisorCall\);
    exception.syndrome<15:0> = immediate;

    if PSTATE.EL == EL3 then
        AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL2, exception, 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();
  SSAdvance();
  bits(64) preferred_exception_return = NextInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_MonitorCall);
  exception.syndrome<15:0> = immediate;

  AArch64.TakeException(EL3, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/syscalls/AArch64.CallSupervisor

```
// AArch64.CallSupervisor()
// =====
// Calls the Supervisor

AArch64.CallSupervisor(bits(16) immediate_in)
  bits(16) immediate = immediate_in;
  if UsingAArch32() then AArch32.ITAdvance();
  SSAdvance();
  route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';

  bits(64) preferred_exception_return = NextInstrAddr();
  vect_offset = 0x0;

  exception = ExceptionSyndrome(Exception_SupervisorCall);
  exception.syndrome<15:0> = immediate;

  if UInt(PSTATE.EL) > UInt(EL1) then
    AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
  elsif route_to_el2 then
    AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
  else
    AArch64.TakeException(EL1, exception, 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);
ExceptionRecord exception = exception_in;
boolean sync_errors;
boolean iesb_req;
if HaveIESB() then
    sync_errors = SCTLR[target_el].IESB == '1';
    if HaveDoubleFaultExt() then
        sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && target_el == EL3);
    if sync_errors && InsertIESBBeforeException(target_el) then
        SynchronizeErrors();
        iesb_req = FALSE;
        sync_errors = FALSE;
        TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
else
    sync_errors = FALSE;

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();

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);
    elsif 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);

if PSTATE.EL == EL1 && target_el == EL1 && EL2Enabled() then
    if HaveNV2Ext() && (HCR_EL2.<NV,NV1,NV2> == '100' || HCR_EL2.<NV,NV1,NV2> == '111') then
        spsr<3:2> = '10';
    else
        if HaveNVExt() && HCR_EL2.<NV,NV1> == '10' then
            spsr<3:2> = '10';

if HaveBTIExt() && !UsingAArch32() then
    boolean zero_btype;
    // SPSR[].BTYP is only guaranteed valid for these exception types
    if exception.exceptype IN {Exception\_SError, Exception\_IRQ0, Exception\_FIQ,
Exception\_SoftwareStep, Exception\_PCAlignment,
Exception\_InstructionAbort, Exception\_Breakpoint,
Exception\_VectorCatch, Exception\_SoftwareBreakpoint,
Exception\_IllegalState, Exception\_BranchTarget} then
        zero_btype = FALSE;
    else
        zero_btype = ConstrainUnpredictableBool(Unpredictable\_ZEROBTYP);
    if zero_btype then spsr<11:10> = '00';

if HaveNV2Ext() && exception.exceptype == Exception\_NV2DataAbort && target_el == EL3 then
    // External aborts are configured to be taken to EL3
    exception.exceptype = Exception\_DataAbort;

```

```

if !(exception.exceptype IN {Exception_IRQ, Exception_FIQ}) then
    AArch64.ReportException(exception, target_el);

PSTATE.EL = target_el;
PSTATE.nRW = '0';
PSTATE.SP = '1';

SPSR[] = spsr;
ELR[] = preferred_exception_return;

PSTATE.SS = '0';
if HaveFeatNMI() && !ELUsingAArch32(target_el) then PSTATE.ALLINT = NOT SCTLR[].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 (HavePANExt() && (PSTATE.EL == EL1 || (PSTATE.EL == EL2 && ELIsInHost(EL0))) &&
    SCTLR[].SPAN == '0') then
    PSTATE.PAN = '1';
if HaveUAOExt() then PSTATE.UAO = '0';
if HaveBTIExt() then PSTATE.BTYPE = '00';
if HaveSSBSExt() then PSTATE.SSBS = SCTLR[].DSSBS;
if HaveMTEEExt() then PSTATE.TCO = '1';

boolean branch_conditional = FALSE;
BranchTo(VBAR[]<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);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = AArch64.AArch32SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);

```



```

// AArch64.AArch32SystemAccessTrapSyndrome()
// =====
// Returns the syndrome information for traps on AArch32 MCR, MCRR, MRC, MRRC, and VMRS, VMSR instructions
// other than traps that are due to HCPTR or CPACR.

ExceptionRecord AArch64.AArch32SystemAccessTrapSyndrome(bits(32) instr, integer ec)
    ExceptionRecord exception;

    case ec of
        when 0x0    exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x3    exception = ExceptionSyndrome(Exception_CP15RTTTrap);
        when 0x4    exception = ExceptionSyndrome(Exception_CP15RRTTTrap);
        when 0x5    exception = ExceptionSyndrome(Exception_CP14RTTTrap);
        when 0x6    exception = ExceptionSyndrome(Exception_CP14DTTTrap);
        when 0x7    exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        when 0x8    exception = ExceptionSyndrome(Exception_FPIDTrap);
        when 0xC    exception = ExceptionSyndrome(Exception_CP14RRTTTrap);
        otherwise   Unreachable();

    bits(20) iss = Zeros();

    if exception.exceptype == Exception_Uncategorized then
        return exception;
    elseif exception.exceptype IN {Exception_FPIDTrap, Exception_CP14RTTTrap, Exception_CP15RRTTTrap} then
        // Trapped MRC/MCR, VMRS on FPSID
        if exception.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 exception.exceptype IN {Exception_CP14RRTTTrap, Exception_AdvSIMDFPAccessTrap, Exception_CP15RRTTTrap} then
        // Trapped MRRC/MCRR, VMRS/VMSR
        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 exception.exceptype == Exception_CP14DTTTrap 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

    exception.syndrome<24:20> = ConditionSyndrome();
    exception.syndrome<19:0> = iss;

    return exception;

```

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)
    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0x0;

    route_to_el2 = (target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1');

    if route_to_el2 then
        exception = ExceptionSyndrome(Exception_Uncategorized);
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
        exception.syndrome<24:20> = ConditionSyndrome();
        AArch64.TakeException(target_el, exception, 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 SCTLR_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 SCTLR_EL1.TIDCP.
    if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && !IsInHost() &&
        !ELUsingAArch32(EL1) && SCTLR_EL1.TIDCP == '1' && trapped_encoding) then
        if EL2Enabled() && HCR_EL2.TGE == '1' then
            AArch64.AArch32SystemAccessTrap(EL2, 0x3);
        else
            AArch64.AArch32SystemAccessTrap(EL1, 0x3);

    // Check for coarse-grained Hyp traps
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then
        // Check for MRC and MCR disabled by SCTLR_EL2.TIDCP.
        if (HaveFeatTIDCP1() && PSTATE.EL == EL0 && IsInHost() &&
            SCTLR_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();
```

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' && EL3SDDTrapPriority() then
    UNDEFINED;

  if PSTATE.EL IN {EL0, EL1, EL2} && EL2Enabled() then
    // Check if access disabled in CPTR_EL2
    if HaveVirtHostExt() && HCR_EL2.E2H == '1' 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 Halted() && EDSCR.SDD == '1' 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() &&
        ((HaveNVExt() && HCR_EL2.NV == '1') ||
        (HaveFGTExt() && (!HaveEL(EL3) || SCR_EL3.FGTEn == '1') &&
        HFGITR_EL2.ERET == '1')));
    if route_to_el2 then
        ExceptionRecord exception;
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_ERetTrap);
        if !eret_with_pac then // ERET
            exception.syndrome<1> = '0';
            exception.syndrome<0> = '0'; // RES0
        else
            exception.syndrome<1> = '1';
            if pac_uses_key_a then // ERETAA
                exception.syndrome<0> = '0';
            else // ERETAB
                exception.syndrome<0> = '1';
        AArch64.TakeException(EL2, exception, 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() then
            if HaveNVExt() && HCR_EL2.NV == '1' && HCR_EL2.TSC == '1' then
                route_to_el2 = TRUE;
            else
                UNDEFINED;
        else
            UNDEFINED;
    else
        route_to_el2 = PSTATE.EL == EL1 && EL2Enabled() && HCR_EL2.TSC == '1';
    if route_to_el2 then
        bits(64) preferred_exception_return = ThisInstrAddr();
        vect_offset = 0x0;
        exception = ExceptionSyndrome(Exception_MonitorCall);
        exception.syndrome<15:0> = imm;
        exception.trappedsyscallinst = TRUE;
        AArch64.TakeException(EL2, exception, 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 HaveFGTExt() then
        route_to_el2 = FALSE;
        if PSTATE.EL == EL0 then
            route_to_el2 = (!UsingAArch32() && !ELUsingAArch32(EL1) &&
                EL2Enabled() && HFGITR_EL2.SVC_EL0 == '1' &&
                (HCR_EL2.<E2H, TGE> != '11' && (!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
            exception = ExceptionSyndrome(Exception_SupervisorCall);
            exception.syndrome<15:0> = immediate;
            exception.trappedsyscallinst = TRUE;
            bits(64) preferred_exception_return = ThisInstrAddr();
            vect_offset = 0x0;

            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.CheckForWFXTrap

```
// AArch64.CheckForWFXTrap()
// =====
// Check for trap on WFE or WFI instruction

AArch64.CheckForWFXTrap(bits(2) target_el, WfxType wfxtype)
    assert HaveEL(target_el);

    boolean is_wfe = wfxtype IN {WfxType_WFE, WfxType_WFET};
    boolean trap;
    case target_el of
        when EL1
            trap = (if is_wfe then SCTLR[].nTWE else SCTLR[].nTWI) == '0';
        when EL2
            trap = (if is_wfe then HCR_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
        AArch64.WFXTrap(wfxtype, 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';

        bits(64) preferred_exception_return = ThisInstrAddr\(\);
        vect_offset = 0x0;

        exception = ExceptionSyndrome\(Exception\_IllegalState\);

        if UInt(PSTATE.EL) > UInt(EL1) then
            AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
        elsif route_to_el2 then
            AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
        else
            AArch64.TakeException(EL1, exception, 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()
    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    exception = ExceptionSyndrome\(Exception\_Uncategorized\);

    if IsSecureEL2Enabled\(\) then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    AArch64.TakeException(EL3, exception, 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);

    bits(64) preferred_exception_return = ThisInstrAddr\(\);
    vect_offset = 0x0;

    exception = AArch64.SystemAccessTrapSyndrome(ThisInstr(), ec);
    AArch64.TakeException(target_el, exception, 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 exception;
    bits(32) instr = instr_in;
    case ec of
        when 0x0 // Trapped access due to unknown reason
            exception = ExceptionSyndrome(Exception_Uncategorized);
        when 0x7 // Trapped access to SVE, Advance SIMD, or SVE2
            exception = ExceptionSyndrome(Exception_AdvSIMDFPAccessTrap);
            exception.syndrome<24:20> = ConditionSyndrome();
        when 0x18 // Trapped access to system register
            exception = ExceptionSyndrome(Exception_SystemRegisterTrap);
            instr = ThisInstr();
            exception.syndrome<21:20> = instr<20:19>; // Op0
            exception.syndrome<19:17> = instr<7:5>; // Op2
            exception.syndrome<16:14> = instr<18:16>; // Op1
            exception.syndrome<13:10> = instr<15:12>; // CRn
            exception.syndrome<9:5> = instr<4:0>; // Rt
            exception.syndrome<4:1> = instr<11:8>; // CRm
            exception.syndrome<0> = instr<21>; // Direction
        otherwise
            Unreachable();

    return exception;
```

Library pseudocode for aarch64/exceptions/traps/AArch64.UndefinedFault

```
// AArch64.UndefinedFault()
// =====

AArch64.UndefinedFault()

    route_to_el2 = PSTATE.EL == EL0 && EL2Enabled() && HCR_EL2.TGE == '1';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_Uncategorized);

    if UInt(PSTATE.EL) > UInt(EL1) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elseif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/exceptions/traps/AArch64.WFxTrap

```
// AArch64.WFxTrap()
// =====

AArch64.WFxTrap(WFxType wfxtype, bits(2) target_el)
    assert UInt(target_el) > UInt(PSTATE.EL);

    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_WFxTrap);
    exception.syndrome<24:20> = ConditionSyndrome();

    case wfxtype of
        when WFxType_WFI
            exception.syndrome<1:0> = '00';
        when WFxType_WFE
            exception.syndrome<1:0> = '01';
        when WFxType_WFIT
            exception.syndrome<1:0> = '10';
            if HaveFeatWFXT2() then
                exception.syndrome<2> = '1'; // Register field is valid
                exception.syndrome<9:5> = ThisInstr()<4:0>;
            else
                exception.syndrome<2> = '0'; // Register field is invalid
        when WFxType_WFET
            exception.syndrome<1:0> = '11';
            if HaveFeatWFXT2() then
                exception.syndrome<2> = '1'; // Register field is valid
                exception.syndrome<9:5> = ThisInstr()<4:0>;
            else
                exception.syndrome<2> = '0'; // Register field is invalid

    if target_el == EL1 && EL2Enabled() && HCR_EL2.TGE == '1' then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(target_el, exception, 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/CheckFPEEnabled64

```
// CheckFPEEnabled64()
// =====
// AArch64 instruction wrapper

CheckFPEEnabled64()
    AArch64.CheckFPEEnabled();
```

Library pseudocode for aarch64/exceptions/traps/CheckLDST64BEnabled

```
// CheckLDST64BEnabled()
// =====
// Checks for trap on ST64B and LD64B instructions

CheckLDST64BEnabled()
  boolean trap = FALSE;
  bits(25) iss = ZeroExtend('10'); // 0x2
  bits(2) target_el;

  if PSTATE.EL == EL0 then
    if !IsInHost() then
      trap = SCTLRL_EL1.EnALS == '0';
      target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
    else
      trap = SCTLRL_EL2.EnALS == '0';
      target_el = EL2;
  else
    target_el = EL1;

  if (!trap && EL2Enabled() && HaveFeatHGX() &&
      ((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;
  bits(25) iss = ZeroExtend('1'); // 0x1
  bits(2) target_el;

  if (PSTATE.EL != EL3 && HaveEL(EL3) &&
      SCR_EL3.EnAS0 == '0' && EL3SDDTrapPriority()) then
    UNDEFINED;

  if PSTATE.EL == EL0 then
    if !IsInHost() then
      trap = SCTLRL_EL1.EnAS0 == '0';
      target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
    else
      trap = SCTLRL_EL2.EnAS0 == '0';
      target_el = EL2;

  if (!trap && EL2Enabled() && HaveFeatHGX() &&
      ((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 && Halted() && EDSCR.SDD == '1' 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;
    bits(25) iss = Zeros();
    bits(2) target_el;

    if PSTATE.EL == EL0 then
        if !IsInHost() then
            trap = SCTL_EL1.EnASR == '0';
            target_el = if EL2Enabled() && HCR_EL2.TGE == '1' then EL2 else EL1;
        else
            trap = SCTL_EL2.EnASR == '0';
            target_el = EL2;

    if (!trap && EL2Enabled() && HaveFeatHCX() &&
        ((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)
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_LDST64BTrap);
    exception.syndrome = iss;
    AArch64.TakeException(target_el, exception, 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

```
// Returns TRUE if WaitForEvent() returns before WFE trap delay expires,
// FALSE otherwise.
boolean WaitForEventUntilDelay(boolean delay_enabled, integer delay);
```

Library pseudocode for aarch64/functions/aborts/AArch64.FaultSyndrome

```
// AArch64.FaultSyndrome()
// =====
// Creates an exception syndrome value for Abort and Watchpoint exceptions taken to
// an Exception level using AArch64.

(bits(25), bits(5)) AArch64.FaultSyndrome(boolean d_side, FaultRecord fault)
    assert fault.statuscode != Fault\_None;

    bits(25) iss = Zeros();
    bits(5) iss2 = Zeros();

    if !HaveFeatLS64() && HaveRASExt() && IsAsyncAbort(fault) then
        iss<12:11> = fault.errortype; // SET

    if d_side then
        if HaveFeatLS64() && fault.acctype == AccType\_ATOMICLS64 then
            if (fault.statuscode IN {Fault\_AccessFlag,
                Fault\_Translation, Fault\_Permission}) then
                (iss2, iss<24:14>, iss<12:11>) = LS64InstructionSyndrome();
            else
                if (IsSecondStage(fault) && !fault.s2fslwalk &&
                    (!IsExternalSyncAbort(fault) ||
                     (!HaveRASExt() && fault.acctype == AccType\_TTW &&
                      boolean IMPLEMENTATION_DEFINED "ISV on second stage translation table walk")))) then
                    iss<24:14> = LSInstructionSyndrome();

                if HaveNV2Ext() && fault.acctype == AccType\_NV2REGISTER then
                    iss<13> = '1'; // Fault is generated by use of VNCR_EL2

                if fault.acctype IN {AccType\_DC, AccType\_IC, AccType\_AT, AccType\_ATPAN} then
                    iss<8> = '1'; iss<6> = '1';
                else
                    iss<6> = if fault.write then '1' else '0';

                if IsExternalAbort(fault) then iss<9> = fault.extflag;
                iss<7> = if fault.s2fslwalk then '1' else '0';
                iss<5:0> = EncodeLDFSC(fault.statuscode, fault.level);

    return (iss, iss2);
```

Library pseudocode for aarch64/functions/aborts/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, and LST.
(bits(5), bits(11), bits(2)) LS64InstructionSyndrome();
```

Library pseudocode for aarch64/functions/cache/AArch64.DataMemZero

```
// AArch64.DataMemZero()
// =====
// Write Zero to data memory

AArch64.DataMemZero(bits(64) regval, bits(64) vaddress, AddressDescriptor memaddrdesc_in, integer size)
    iswrite = TRUE;
    AddressDescriptor memaddrdesc = memaddrdesc_in;
    for i = 0 to size-1
        accdesc = CreateAccessDescriptor(AccType\_DCZVA);
        if HaveMTEExt() then
            if AArch64.AccessIsTagChecked(vaddress, AccType\_DCZVA) then
                bits(4) ptag = AArch64.PhysicalTag(vaddress);
                if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                    if boolean IMPLEMENTATION_DEFINED "DC_ZVA tag fault reported with lowest faulting address" then
                        AArch64.TagCheckFault(vaddress, AccType\_DCZVA, iswrite);
                    else
                        AArch64.TagCheckFault(regval, AccType\_DCZVA, iswrite);
                memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, Zeros());
                if IsFault(memstatus) then
                    HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
            memaddrdesc.address.address = memaddrdesc.address.address + 1;
    return;
```

Library pseudocode for aarch64/functions/cache/AArch64.TagMemZero

```
// AArch64.TagMemZero()
// =====
// Write Zero to tag memory

AArch64.TagMemZero(bits(64) vaddress_in, integer size)
    bits(64) vaddress = vaddress_in;
    integer count = size >> LOG2\_TAG\_GRANULE;
    bits(4) tag = AArch64.AllocationTagFromAddress(vaddress);
    for i = 0 to count-1
        AArch64.MemTag[vaddress, AccType\_NORMAL] = tag;
        vaddress = vaddress + TAG\_GRANULE;
    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.

boolean AArch64.ExclusiveMonitorsPass(bits(64) address, integer size)

    // 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.

    acctype = AccType\_ATOMIC;
    iswrite = TRUE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    passed = AArch64.IsExclusiveVA(address, ProcessorID(), size);
    if !passed then
        return FALSE;

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    passed = IsExclusiveLocal(memaddrdesc.address, ProcessorID(), size);
    ClearExclusiveLocal(ProcessorID());

    if passed then
        if memaddrdesc.memattrs.shareability != Shareability\_NSH then
            passed = IsExclusiveGlobal(memaddrdesc.address, ProcessorID(), size);

    return passed;
```

Library pseudocode for aarch64/functions/exclusive/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

```
// 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)
    acctype = AccType\_ATOMIC;
    iswrite = FALSE;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, 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/fusedrstep/FPRsqrtStepFused

```
// FPRsqrtStepFused()
// =====

bits(N) FPRsqrtStepFused(bits(N) op1_in, bits(N) op2)
    assert N IN {16, 32, 64};
    bits(N) result;
    bits(N) op1 = op1_in;
    boolean done;
    FPCRType fpcr = FPCR[];
    op1 = FPNeg(op1);
    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    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, FALSE, fpexc);
    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');
        elsif inf1 || inf2 then
            result = FPIfinity(sign1 EOR sign2);
        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);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

    return result;
```

Library pseudocode for aarch64/functions/fusedrstep/FPRecipStepFused

```
// FPRecipStepFused()
// =====

bits(N) FPRecipStepFused(bits(N) op1_in, bits(N) op2)
  assert N IN {16, 32, 64};
  bits(N) op1 = op1_in;
  bits(N) result;
  boolean done;
  FPCRType fpcr = FPCR[];
  op1 = FPNeg(op1);

  boolean altfp = HaveAltFP() && fpcr.AH == '1';
  boolean fpxc = !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, fpxc);
  (type2,sign2,value2) = FPUnpack(op2, fpcr, fpxc);
  (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, FALSE, fpxc);
  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');
    elsif inf1 || inf2 then
      result = FPInfinity(sign1 EOR sign2);
    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);
      else
        result = FPRound(result_value, fpcr, rounding, fpxc);

  return result;
```

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, AccType acctype)
    if PSTATE.M<4> == '1' then return FALSE;

    if EffectiveTBI(vaddr, FALSE, PSTATE.EL) == '0' then
        return FALSE;

    if EffectiveTCMA(vaddr, PSTATE.EL) == '1' && (vaddr<59:55> == '00000' || vaddr<59:55> == '11111') then
        return FALSE;

    if !AArch64.AllocationTagAccessIsEnabled(acctype) then
        return FALSE;

    if acctype IN {AccType\_IFETCH, AccType\_TTW, AccType\_DC, AccType\_IC} then
        return FALSE;

    if acctype == AccType\_NV2REGISTER then
        return FALSE;

    if PSTATE.TC0=='1' then
        return FALSE;

    if !IsTagCheckedInstruction() 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, AccType acctype, bits(4) allocation_tag)
    bits(64) result = address;
    bits(4) tag;
    if AArch64.AllocationTagAccessIsEnabled(acctype) then
        tag = allocation_tag;
    else
        tag = '0000';
    result<59:56> = tag;
    return result;
```

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.CheckAlignment

```
// AArch64.CheckAlignment()
// =====

boolean AArch64.CheckAlignment(bits(64) address, integer alignment, AccType acctype,
                               boolean iswrite)

    aligned = (address == Align(address, alignment));
    atomic  = acctype IN { AccType\_ATOMIC, AccType\_ATOMICRW, AccType\_ORDEREDATOMIC,
                          AccType\_ORDEREDATOMICRW, AccType\_ATOMICS64, AccType\_A32LSMD};
    ordered = acctype IN { AccType\_ORDERED, AccType\_ORDEREDRW, AccType\_LIMITEDORDERED,
                          AccType\_ORDEREDATOMIC, AccType\_ORDEREDATOMICRW };
    vector  = acctype == AccType\_VEC;
    boolean check;
    if SCTLR[].A == '1' then check = TRUE;
    elsif HaveLSE2Ext() then
        check = (UInt(address<3:0>) + alignment > 16) && ((ordered && SCTLR[].nAA == '0') || atomic);
    else check = atomic || ordered;

    if check && !aligned then
        secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

    return aligned;
```

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) ptag, boolean v)
    if memaddrdesc.memattrs.tagged then
        (memstatus, readtag) = PhysMemTagRead(memaddrdesc, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        return ptag == readtag;
    else
        return TRUE;
```



```

// 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, AccType acctype, boolean aligned]
    boolean ispair = FALSE;
    return AArch64.MemSingle[address, size, acctype, aligned, ispair];

// 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, AccType acctype, boolean aligned, boolean
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    if HaveLSE2Ext() then
        assert CheckAllInAlignedQuantity(address, size, 16);
    else
        assert address == Align(address, size);

AddressDescriptor memaddrdesc;
bits(size*8) value;
iswrite = FALSE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite);
PhysMemRetStatus memstatus;
if atomic then
    (memstatus, value) = PhysMemRead(memaddrdesc, size, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
elseif splitpair then
    assert ispair;
    bits(halfsize * 8) lowhalf, highhalf;
    (memstatus, lowhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    (memstatus, highhalf) = PhysMemRead(memaddrdesc, halfsize, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, halfsize, accdesc);

    value = highhalf:lowhalf;
else
    for i = 0 to size-1
        (memstatus, value<8*i+7:8*i>) = PhysMemRead(memaddrdesc, 1, accdesc);
        if IsFault(memstatus) then
            HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return value;

// AArch64.MemSingle[] - assignment (write) form
// =====

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned] = bits(size*8) value
    boolean ispair = FALSE;
    AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;

```

```

return;

// AArch64.MemSingle[] - assignment (write) form
// =====
// Perform an atomic, little-endian write of 'size' bytes.

AArch64.MemSingle[bits(64) address, integer size, AccType acctype, boolean aligned, boolean ispair] = bit
assert size IN {1, 2, 4, 8, 16};
constant halfsize = size DIV 2;
if HaveLSE2Ext() then
    assert CheckAllInAlignedQuantity(address, size, 16);
else
    assert address == Align(address, size);

AddressDescriptor memaddrdesc;
iswrite = TRUE;

memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Effect on exclusives
if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), size);

// Memory array access
accdesc = CreateAccessDescriptor(acctype);

if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
        bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(ZeroExtend(address, 64), acctype, iswrite);

PhysMemRetStatus memstatus;
(atomic, splitpair) = CheckSingleAccessAttributes(address, memaddrdesc.memattrs, size, acctype, iswrite);
if atomic then
    memstatus = PhysMemWrite(memaddrdesc, size, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, size, accdesc);
elseif splitpair then
    assert ispair;
    bits(halfsize*8) lowhalf, highhalf;
    <highhalf, lowhalf> = value;

    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, lowhalf);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
    memaddrdesc.paddress.address = memaddrdesc.paddress.address + halfsize;
    memstatus = PhysMemWrite(memaddrdesc, halfsize, accdesc, highhalf);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, halfsize, accdesc);
else
    for i = 0 to size-1
        memstatus = PhysMemWrite(memaddrdesc, 1, accdesc, value<8*i+7:8*i>);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
        memaddrdesc.paddress.address = memaddrdesc.paddress.address + 1;
return;

```

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, AccType acctype]
AddressDescriptor memaddrdesc;
bits(4) value;

iswrite = FALSE;
aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
                                       TAG\_GRANULE);
accdesc = CreateAccessDescriptor(acctype);
// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

// Return the granule tag if tagging is enabled...
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    (memstatus, tag) = PhysMemTagRead(memaddrdesc, accdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, 1, accdesc);
    return tag;
else
    // ...otherwise read tag as zero.
    return '0000';

// AArch64.MemTag[] - assignment (write) form
// =====
// Store an Allocation Tag to memory.

AArch64.MemTag[bits(64) address, AccType acctype] = bits(4) value
AddressDescriptor memaddrdesc;
iswrite = TRUE;

// Stores of allocation tags must be aligned
if address != Align(address, TAG\_GRANULE) then
    boolean secondstage = FALSE;
    AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

aligned = TRUE;
memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned,
                                       TAG\_GRANULE);

// It is CONSTRAINED UNPREDICTABLE if tags stored to memory locations marked as Device
// generate an Alignment Fault or store the data to locations.
if memaddrdesc.memattrs.memtype == MemType\_Device then
    c = ConstrainUnpredictable(Unpredictable\_DEVICETAGSTORE);
    assert c IN {Constraint\_NONE, Constraint\_FAULT};
    if c == Constraint\_FAULT then
        boolean secondstage = FALSE;
        AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));

// Check for aborts or debug exceptions
if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

accdesc = CreateAccessDescriptor(acctype);
// Memory array access
if AArch64.AllocationTagAccessIsEnabled(acctype) && memaddrdesc.memattrs.tagged then
    memstatus = PhysMemTagWrite(memaddrdesc, accdesc, value);
    if IsFault(memstatus) then
        HandleExternalWriteAbort(memstatus, memaddrdesc, 1, accdesc);
```

Library pseudocode for aarch64/functions/memory/AArch64.PhysicalTag

```
// AArch64.PhysicalTag()
// =====
// Generate a Physical Tag from a Logical Tag in an address

bits(4) AArch64.PhysicalTag(bits(64) vaddr)
    return vaddr<59:56>;
```

Library pseudocode for aarch64/functions/memory/AArch64.TranslateAddressForAtomicAccess

```
// AArch64.TranslateAddressForAtomicAccess()
// =====
// Performs an alignment check for atomic memory operations.
// Also translates 64-bit Virtual Address into Physical Address.

AddressDescriptor AArch64.TranslateAddressForAtomicAccess(bits(64) address, integer sizeinbits)
    boolean iswrite = FALSE;
    size = sizeinbits DIV 8;

    assert size IN {1, 2, 4, 8, 16};

    aligned = AArch64.CheckAlignment(address, size, AccType\_ATOMICRW, iswrite);

    // MMU or MPU lookup
    memaddrdesc = AArch64.TranslateAddress(address, AccType\_ATOMICRW, iswrite,
                                           aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.address, ProcessorID(), size);

    if HaveMTE2Ext() && AArch64.AccessIsTagChecked(address, AccType\_ATOMICRW) then
        bits(4) ptag = AArch64.PhysicalTag(address);
        accdesc = CreateAccessDescriptor(AccType\_ATOMICRW);
        if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
            AArch64.TagCheckFault(address, AccType\_ATOMICRW, iswrite);

    return memaddrdesc;
```

Library pseudocode for aarch64/functions/memory/AddressSupportsLS64

```
// Returns TRUE if the 64-byte block following the given address supports the
// LD64B and ST64B instructions, and FALSE otherwise.
boolean AddressSupportsLS64(bits(64) address);
```

Library pseudocode for aarch64/functions/memory/CheckAllInAlignedQuantity

```
// CheckAllInAlignedQuantity()
// =====
// Returns TRUE if all accessed bytes are within one aligned quantity, FALSE otherwise.

boolean CheckAllInAlignedQuantity(bits(64) address, integer size, integer alignment)
    assert(size <= alignment);
    return Align(address+size-1, alignment) == Align(address, alignment);
```

Library pseudocode for aarch64/functions/memory/CheckSPAlignment

```
// CheckSPAlignment()
// =====
// Check correct stack pointer alignment for AArch64 state.

CheckSPAlignment()
  bits(64) sp = SP[];
  boolean stack_align_check;
  if PSTATE.EL == EL0 then
    stack_align_check = (SCTLR[].SA0 != '0');
  else
    stack_align_check = (SCTLR[].SA != '0');

  if stack_align_check && sp != Align(sp, 16) then
    AArch64.SPAlignmentFault();

  return;
```

Library pseudocode for aarch64/functions/memory/CheckSingleAccessAttributes

```
// CheckSingleAccessAttributes()
// =====
//
// When FEAT_LSE2 is implemented, a MemSingle[] access needs to be further assessed once the memory
// attributes are determined.
// If it was aligned to access size or targets Normal Inner Write-Back, Outer Write-Back Cacheable
// memory then it is single copy atomic and there is no alignment fault.
// If not, for exclusives, atomics and non atomic acquire release instructions - it is CONSTRAINED UNPREDICTABLE
// if they generate an alignment fault. If they do not generate an alignment fault - they are
// single copy atomic.
// Otherwise it is IMPLEMENTATION DEFINED - if they are single copy atomic.
//
// The function returns (atomic, splitpair), where
// atomic indicates if the access is single copy atomic.
// splitpair indicates that a load/store pair is split into 2 single copy atomic accesses.
// when atomic and splitpair are both FALSE - the access is not single copy atomic and may be treated
// as byte accesses.

(boolean, boolean) CheckSingleAccessAttributes(bits(64) address, MemoryAttributes memattrs, integer size,
AccType acctype, boolean iswrite, boolean aligned, boolean ispair)
    isnormalwb = (memattrs.memtype == MemType\_Normal &&
        memattrs.inner.attrs == MemAttr\_WB &&
        memattrs.outer.attrs == MemAttr\_WB);

    atomic = TRUE;
    splitpair = FALSE;
    if isnormalwb then return (atomic, splitpair);

    accatomic = acctype IN { AccType\_ATOMIC, AccType\_ATOMICRW, AccType\_ORDEREDATOMIC,
        AccType\_ORDEREDATOMICRW, AccType\_ATOMICLS64, AccType\_A32LSMD};
    ordered = acctype IN { AccType\_ORDERED, AccType\_ORDEREDRW, AccType\_LIMITEDORDERED, AccType\_ORDEREDATOMIC};

    if !aligned && (accatomic || ordered) then
        atomic = ConstrainUnpredictableBool(Unpredictable\_MISALIGNEDATOMIC);
        if !atomic then
            secondstage = FALSE;
            AArch64.Abort(address, AlignmentFault(acctype, iswrite, secondstage));
        else
            return (atomic, splitpair);

    if ispair && aligned then
        // load / store pair requests that are aligned to each register access are split into 2 single copy atomic
        atomic = FALSE;
        splitpair = TRUE;
        return (atomic, splitpair);

    if aligned then
        return (atomic, splitpair);

    atomic = boolean IMPLEMENTATION_DEFINED "Misaligned accesses within 16 byte aligned memory but not No
    return (atomic, splitpair);
```

Library pseudocode for aarch64/functions/memory/IsTagCheckedInstruction

```
// Returns True if the current instruction uses tag-checked memory access,
// False otherwise.
boolean IsTagCheckedInstruction();
```



```

// 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, AccType acctype]
    boolean ispair = FALSE;
    return Mem[address, size, acctype, ispair];

bits(size*8) Mem[bits(64) address, integer size, AccType acctype, boolean ispair]
    assert size IN {1, 2, 4, 8, 16};
    constant halfsize = size DIV 2;
    bits(size * 8) value;
    bits(halfsize * 8) lowhalf, highhalf;
    boolean iswrite = FALSE;
    boolean aligned;
    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    boolean atomic;
    if size != 16 || !(acctype IN {AccType_VEC, AccType_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elsif acctype IN {AccType_VEC, AccType_VECSTREAM} then
        // 128-bit SIMD&FP loads are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_pair = FALSE;
        single_is_aligned = TRUE;
        lowhalf = AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, single_is_pair];
        highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, single_is_
        value = highhalf:lowhalf;
    elsif atomic && ispair then
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];
    elsif !atomic then

        assert size > 1;
        value<7:0> = AArch64.MemSingle[address, 1, acctype, aligned];

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        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
            value<8*i+7:8*i> = AArch64.MemSingle[address+i, 1, acctype, aligned];
    elsif size == 16 && acctype IN {AccType_VEC, AccType_VECSTREAM} then
        lowhalf = AArch64.MemSingle[address, halfsize, acctype, aligned, ispair];
        highhalf = AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair];
        value = highhalf:lowhalf;
    else
        value = AArch64.MemSingle[address, size, acctype, aligned, ispair];

    if BigEndian(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, AccType acctype] = bits(size*8) value_in
    boolean ispair = FALSE;
    Mem[address, size, acctype, ispair] = value_in;

Mem[bits(64) address, integer size, AccType acctype, boolean ispair] = bits(size*8) value_in
    boolean iswrite = TRUE;
    constant halfsize = size DIV 2;
    bits(size*8) value = value_in;
    bits(halfsize*8) lowhalf, highhalf;
    boolean atomic;
    boolean aligned;
    if BigEndian(acctype) then
        value = BigEndianReverse(value);

    if ispair then
        // check alignment on size of element accessed, not overall access size
        aligned = AArch64.CheckAlignment(address, halfsize, acctype, iswrite);
    else
        aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    if ispair then
        atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif size != 16 || !(acctype IN {AccType\_VEC, AccType\_VECSTREAM}) then
        if !HaveLSE2Ext() then
            atomic = aligned;
        else
            atomic = CheckAllInAlignedQuantity(address, size, 16);
    elseif (acctype IN {AccType\_VEC, AccType\_VECSTREAM}) then
        // 128-bit SIMD&FP stores are treated as a pair of 64-bit single-copy atomic accesses
        // 64-bit aligned.
        atomic = address == Align(address, 8);
    else
        // 16-byte integer access
        atomic = address == Align(address, 16);

    if !atomic && ispair && address == Align(address, halfsize) then
        single_is_aligned = TRUE;
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, single_is_aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, single_is_aligned, ispair] = highhalf;
    elseif atomic && ispair then
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
    elseif !atomic then
        assert size > 1;
        AArch64.MemSingle[address, 1, acctype, aligned] = value<7:0>;

        // For subsequent bytes it is CONSTRAINED UNPREDICTABLE whether an unaligned Device memory
        // access will generate an Alignment Fault, as to get this far means the first byte did
        // not, so we must be changing to a new translation page.
        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
            AArch64.MemSingle[address+i, 1, acctype, aligned] = value<8*i+7:8*i>;
    elseif size == 16 && acctype IN {AccType\_VEC, AccType\_VECSTREAM} then
        <highhalf, lowhalf> = value;
        AArch64.MemSingle[address, halfsize, acctype, aligned, ispair] = lowhalf;
        AArch64.MemSingle[address + halfsize, halfsize, acctype, aligned, ispair] = highhalf;
    else
        AArch64.MemSingle[address, size, acctype, aligned, ispair] = value;
return;

```

Library pseudocode for aarch64/functions/memory/MemAtomic

```
// MemAtomic()
// =====
// Performs load and store memory operations for a given virtual address.

bits(size) MemAtomic(bits(64) address, MemAtomicOp op, bits(size) value, AccType ldacctype, AccType stacctype,
bits(size) newvalue;
memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
ldaccdesc = CreateAccessDescriptor(ldacctype);
staccdesc = CreateAccessDescriptor(stacctype);

// All observers in the shareability domain observe the
// following load and store atomically.
(memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
if BigEndian(ldacctype) then
    oldvalue = BigEndianReverse(oldvalue);

case op of
when MemAtomicOp_ADD    newvalue = oldvalue + value;
when MemAtomicOp_BIC    newvalue = oldvalue AND NOT(value);
when MemAtomicOp_EOR    newvalue = oldvalue EOR value;
when MemAtomicOp_ORR    newvalue = oldvalue OR value;
when MemAtomicOp_SMAX   newvalue = if SInt(oldvalue) > SInt(value) then oldvalue else value;
when MemAtomicOp_SMIN   newvalue = if SInt(oldvalue) > SInt(value) then value else oldvalue;
when MemAtomicOp_UMAX   newvalue = if UInt(oldvalue) > UInt(value) then oldvalue else value;
when MemAtomicOp_UMIN   newvalue = if UInt(oldvalue) > UInt(value) then value else oldvalue;
when MemAtomicOp_SWP    newvalue = value;

if BigEndian(stacctype) then
    newvalue = BigEndianReverse(newvalue);
memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
if IsFault(memstatus) then
    HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);

// Load operations return the old (pre-operation) value
return oldvalue;
```

Library pseudocode for aarch64/functions/memory/MemAtomicCompareAndSwap

```
// MemAtomicCompareAndSwap()
// =====
// Compares the value stored at the passed-in memory address against the passed-in expected
// value. If the comparison is successful, the value at the passed-in memory address is swapped
// with the passed-in new_value.

bits(size) MemAtomicCompareAndSwap(bits(64) address, bits(size) expectedvalue,
                                   bits(size) newvalue_in, AccType ldacctype, AccType stacctype)
    bits(size) newvalue = newvalue_in;
    memaddrdesc = AArch64.TranslateAddressForAtomicAccess(address, size);
    ldaccdesc = CreateAccessDescriptor(ldacctype);
    staccdesc = CreateAccessDescriptor(stacctype);

    // All observers in the shareability domain observe the
    // following load and store atomically.
    (memstatus, oldvalue) = PhysMemRead(memaddrdesc, size DIV 8, ldaccdesc);
    if IsFault(memstatus) then
        HandleExternalReadAbort(memstatus, memaddrdesc, size DIV 8, ldaccdesc);
    if BigEndian(ldacctype) then
        oldvalue = BigEndianReverse(oldvalue);

    if oldvalue == expectedvalue then
        if BigEndian(stacctype) then
            newvalue = BigEndianReverse(newvalue);
        memstatus = PhysMemWrite(memaddrdesc, size DIV 8, staccdesc, newvalue);
        if IsFault(memstatus) then
            HandleExternalWriteAbort(memstatus, memaddrdesc, size DIV 8, staccdesc);
    return oldvalue;
```

Library pseudocode for aarch64/functions/memory/MemLoad64B

```
// MemLoad64B()
// =====
// Performs an atomic 64-byte read from a given virtual address.

bits(512) MemLoad64B(bits(64) address, AccType acctype)
  bits(512) data;
  boolean iswrite = FALSE;
  constant integer size = 64;

  aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

  if !AddressSupportsLS64(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.
      boolean secondstage = FALSE;
      boolean s2fslwalk = FALSE;
      FaultRecord fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
      AArch64.Abort(address, fault);
    else
      // Accesses are not single-copy atomic above the byte level
      for i = 0 to 63
        data<7+8*i : 8*i> = AArch64.MemSingle[address+8*i, 1, acctype, aligned];
      return data;

  AddressDescriptor memaddrdesc;
  memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

  // Check for aborts or debug exceptions
  if IsFault(memaddrdesc) then
    AArch64.Abort(address, memaddrdesc.fault);

  // Effect on exclusives
  if memaddrdesc.memattrs.shareability != Shareability\_NSH then
    ClearExclusiveByAddress(memaddrdesc.address, ProcessorID(), size);

  // Memory array access
  accdesc = CreateAccessDescriptor(acctype);

  if HaveMTE2Ext() then
    if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
      bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
      if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
        AArch64.TagCheckFault(address, acctype, iswrite);

  PhysMemRetStatus memstatus;
  (memstatus, data) = PhysMemRead(memaddrdesc, size, accdesc);
  if IsFault(memstatus) then
    HandleExternalReadAbort(memstatus, memaddrdesc, size, accdesc);
  return data;
```

Library pseudocode for aarch64/functions/memory/MemStore64B

```
// 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, AccType acctype)
    boolean iswrite = TRUE;
    constant integer size = 64;
    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);

    if !AddressSupportsLS64(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.
            boolean secondstage = FALSE;
            boolean s2fslwalk = FALSE;
            fault = AArch64.ExclusiveFault(acctype, iswrite, secondstage, s2fslwalk);
            AArch64.Abort(address, fault);
        else
            // Accesses are not single-copy atomic above the byte level.
            for i = 0 to 63
                AArch64.MemSingle[address+8*i, 1, acctype, aligned] = value<7+8*i : 8*i>;
    else
        -= MemStore64BWithRet(address, value, acctype); // Return status is ignored by ST64B
    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, AccType acctype)
    AddressDescriptor memaddrdesc;
    boolean iswrite = TRUE;
    constant integer size = 64;

    aligned = AArch64.CheckAlignment(address, size, acctype, iswrite);
    memaddrdesc = AArch64.TranslateAddress(address, acctype, iswrite, aligned, size);

    // Check for aborts or debug exceptions
    if IsFault(memaddrdesc) then
        AArch64.Abort(address, memaddrdesc.fault);
        return ZeroExtend('1');

    // Effect on exclusives
    if memaddrdesc.memattrs.shareability != Shareability\_NSH then
        ClearExclusiveByAddress(memaddrdesc.paddress, ProcessorID(), 64);

    // Memory array access
    accdesc = CreateAccessDescriptor(acctype);

    if HaveMTE2Ext() then
        if AArch64.AccessIsTagChecked(ZeroExtend(address, 64), acctype) then
            bits(4) ptag = AArch64.PhysicalTag(ZeroExtend(address, 64));
            if !AArch64.CheckTag(memaddrdesc, accdesc, ptag, iswrite) then
                AArch64.TagCheckFault(address, acctype, iswrite);
                return ZeroExtend('1');

    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

```
// 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;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    return Mem[address, 8, AccType_NV2REGISTER];

// 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;
    bits(64) address = SignExtend(VNCR_EL2.BADDR:offset<11:0>, 64);
    Mem[address, 8, AccType_NV2REGISTER] = value;
    return;
```

Library pseudocode for aarch64/functions/memory/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.address 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

```
// 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.address 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/SetTagCheckedInstruction

```
// Flag the current instruction as using/not using memory tag checking.
SetTagCheckedInstruction(boolean checked);
```

Library pseudocode for aarch64/functions/mops/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.
// Postsize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPostSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/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.
// Presize is encoded as -1*size for an option A implementation if cpysize is negative.
bits(64) CPYPreSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/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.
integer CPYSizeChoice(bits(64) toaddress, bits(64) fromaddress, bits(64) cpysize);
```

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() &&
        (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0') &&
        (!IsHCRXEL2Enabled() || HCRX_EL2.MSCEn == '0')) then
        UNDEFINED;

    if (PSTATE.EL == EL0 && SCTLR_EL1.MSCEn == '0' &&
        (!EL2Enabled() || HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0')) then
        UNDEFINED;

    if PSTATE.EL == EL0 && IsInHost() && SCTLR_EL2.MSCEn == '0' then
        UNDEFINED;
```

Library pseudocode for aarch64/functions/mops/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";
```

Library pseudocode for aarch64/functions/mops/MemCpyAccessTypes

```
// MemCpyAccessTypes()
// =====
// Return the read and write access types for a CPY* instruction.

(AccType, AccType) MemCpyAccessTypes(bits(4) options)
    unpriv_at_el1 = PSTATE.EL == EL1 && !(EL2Enabled\(\) &&
        HaveNVExt\(\) && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = PSTATE.EL == EL2 && HaveVirtHostExt\(\) && HCR_EL2.<E2H,TGE> == '11';

    runpriv_at_el1 = options<1> == '1' && unpriv_at_el1;
    runpriv_at_el2 = options<1> == '1' && unpriv_at_el2;
    wunpriv_at_el1 = options<0> == '1' && unpriv_at_el1;
    wunpriv_at_el2 = options<0> == '1' && unpriv_at_el2;

    user_access_override = HaveUA0Ext\(\) && PSTATE.UA0 == '1';

    AccType racctype;
    if !user_access_override && (runpriv_at_el1 || runpriv_at_el2) then
        racctype = if options<3> == '0' then AccType\_UNPRIV else AccType\_UNPRIVSTREAM;
    else
        racctype = if options<3> == '0' then AccType\_NORMAL else AccType\_STREAM;

    AccType wacctype;
    if !user_access_override && (wunpriv_at_el1 || wunpriv_at_el2) then
        wacctype = if options<2> == '0' then AccType\_UNPRIV else AccType\_UNPRIVSTREAM;
    else
        wacctype = if options<2> == '0' then AccType\_NORMAL else AccType\_STREAM;

    return (racctype, wacctype);
```

Library pseudocode for aarch64/functions/mops/MemCpyDirectionChoice

```
// Returns true if in the non-overlapping case of 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 MemCpyDirectionChoice(bits(64) fromaddress, bits(64) toaddress, bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyOptionA

```
// MemCpyOptionA()
// =====
// Returns TRUE if the implementation uses Option A for the
// CPY*/SET* instructions, and FALSE otherwise.

boolean MemCpyOptionA()
    return boolean IMPLEMENTATION_DEFINED "CPY*/SET* instructions use Option A";
```

Library pseudocode for aarch64/functions/mops/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(bits(64) toaddress, bits(64) fromaddress,
    bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/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(bits(64) toaddress, bits(64) fromaddress,
    bits(64) cpysize);
```

Library pseudocode for aarch64/functions/mops/MemCpyZeroSizeCheck

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemCpyZeroSizeCheck();
```

Library pseudocode for aarch64/functions/mops/MemSetAccessType

```
// MemSetAccessType()
// =====
// Return the access type for a SET* instruction.

AccType MemSetAccessType(bits(2) options)
    unpriv_at_el1 = options<0> == '1' && PSTATE.EL == EL1 && !(EL2Enabled() &&
        HaveNVExt() && HCR_EL2.<NV,NV1> == '11');
    unpriv_at_el2 = (options<0> == '1' && PSTATE.EL == EL2 &&
        HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11');

    user_access_override = HaveUA0Ext() && PSTATE.UA0 == '1';

    AccType acctype;
    if !user_access_override && (unpriv_at_el1 || unpriv_at_el2) then
        acctype = if options<1> == '0' then AccType_UNPRIV else AccType_UNPRIVSTREAM;
    else
        acctype = if options<1> == '0' then AccType_NORMAL else AccType_STREAM;

    return acctype;
```

Library pseudocode for aarch64/functions/mops/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(bits(64) toaddress, bits(64) setsize,
    boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/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(bits(64) toaddress, bits(64) setsize,
    boolean IsSETGM);
```

Library pseudocode for aarch64/functions/mops/MemSetZeroSizeCheck

```
// Returns TRUE if the implementation option is checked on a copy of size zero remaining.
boolean MemSetZeroSizeCheck();
```

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(boolean option_a, integer destreg, integer srcreg, integer sizereg,
    boolean wrong_option, boolean from_epilogue, bits(4) options)
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24> = '0';
    exception.syndrome<23> = '0';
    exception.syndrome<22:19> = options;
    exception.syndrome<18> = if from_epilogue then '1' else '0';
    exception.syndrome<17> = if wrong_option then '1' else '0';
    exception.syndrome<16> = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5> = srcreg<4:0>;
    exception.syndrome<4:0> = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, 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(boolean option_a, integer destreg, integer datareg, integer sizereg,
                          boolean wrong_option, boolean from_epilogue, bits(2) options,
                          boolean is_SETG)
    bits(64) preferred_exception_return = ThisInstrAddr();
    integer vect_offset = 0x0;
    bits(2) target_el = MismatchedCpySetTargetEL();

    ExceptionRecord exception = ExceptionSyndrome(Exception_MemCpyMemSet);
    exception.syndrome<24>     = '1';
    exception.syndrome<23>     = if is_SETG then '1' else '0';
    // exception.syndrome<22:21> is RES0
    exception.syndrome<20:19> = options;
    exception.syndrome<18>     = if from_epilogue then '1' else '0';
    exception.syndrome<17>     = if wrong_option then '1' else '0';
    exception.syndrome<16>     = if option_a then '1' else '0';
    // exception.syndrome<15> is RES0
    exception.syndrome<14:10> = destreg<4:0>;
    exception.syndrome<9:5>   = datareg<4:0>;
    exception.syndrome<4:0>   = sizereg<4:0>;

    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

Library pseudocode for aarch64/functions/mops/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.
// Postsize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPostSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGE);
```

Library pseudocode for aarch64/functions/mops/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.
// Presize is encoded as -1*size for an option A implementation if setsize is negative.
bits(64) SETPreSizeChoice(bits(64) toaddress, bits(64) setsize, boolean IsSETGP);
```

Library pseudocode for aarch64/functions/mops/SETSizeChoice

```
// Returns the size of the block thisperformed 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.
integer SETSizeChoice(bits(64) toaddress, bits(64) setsize, integer AlignSize);
```



```

// 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)
    bits(64) PAC;
    bits(64) result;
    bits(64) ext_ptr;
    bits(64) extfield;
    bit selbit;
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer top_bit = if tbi then 55 else 63;

    // 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 HaveEnhancedPAC2() && ConstPACField() then selbit = ptr<55>;
    integer bottom_PAC_bit = CalculateBottomPACBit(selbit);

    // The pointer authentication code field takes all the available bits in between
    extfield = Replicate(selbit, 64);

    // Compute the pointer authentication code for a ptr with good extension bits
    if tbi then
        ext_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        ext_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(ext_ptr, modifier, K<127:64>, K<63:0>);

    // Check if the ptr has good extension bits and corrupt the pointer authentication code if not
    if !IsZero(ptr<top_bit:bottom_PAC_bit>) && !IsOnes(ptr<top_bit:bottom_PAC_bit>) then
        if HaveEnhancedPAC() then
            PAC = 0x0000000000000000<63:0>;
        elseif !HaveEnhancedPAC2() then
            PAC<top_bit-1> = NOT(PAC<top_bit-1>);

```

```

// preserve the determination between upper and lower address at bit<55> and insert PAC
if !HaveEnhancedPAC2() then
    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>;
else
    if tbi then
        result = ptr<63:56>:selbit:(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>):selbit:(ptr<54:bottom_PAC_bit> EOR
            PAC<54:bottom_PAC_bit>):ptr<bottom_PAC_bit-1:0>;
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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDA else SCTL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            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 X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    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;
    boolean TrapEL3;
    bits(128) APGAKey_EL1;

    APGAKey_EL1 = APGAKeyHi_EL1<63:0> : APGAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            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 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return ComputePAC(X, Y, APGAKey_EL1<127:64>, APGAKey_EL1<63:0><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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0>:APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnIA else SCTL_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnIA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnIA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return AddPAC(X, Y, 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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime\(\) == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIB else SCTLR_EL2.EnIB;
            TrapEL2 = (EL2Enabled\(\) && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            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 X;
    elsif TrapEL3 && EL3SDDTrapPriority\(\) then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse\(EL2\);
    elsif TrapEL3 then
        if Halted\(\) && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse\(EL3\);
    else
        return AddPAC\(X, Y, APIBKey\_EL1, FALSE\);
```

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';
    bits(64) preferred_exception_return = ThisInstrAddr();
    vect_offset = 0x0;

    exception = ExceptionSyndrome(Exception_PACFail);
    exception.syndrome<1:0> = syndrome;
    exception.syndrome<24:2> = Zeros(); // RES0

    if UInt(PSTATE.EL) > UInt(EL0) then
        AArch64.TakeException(PSTATE.EL, exception, preferred_exception_return, vect_offset);
    elsif route_to_el2 then
        AArch64.TakeException(EL2, exception, preferred_exception_return, vect_offset);
    else
        AArch64.TakeException(EL1, exception, 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)
    bits(64) PAC;
    bits(64) result;
    bits(64) original_ptr;
    bits(2) error_code;
    bits(64) extfield;

    // Reconstruct the extension field used of adding the PAC to the pointer
    boolean tbi = EffectiveTBI(ptr, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(ptr<55>);
    extfield = Replicate(ptr<55>, 64);

    if tbi then
        original_ptr = ptr<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>:ptr<bottom_PAC_bit-1:0>;

    PAC = ComputePAC(original_ptr, modifier, K<127:64>, K<63:0>);
    // Check pointer authentication code
    if tbi then
        if !HaveEnhancedPAC2() then
            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;
            result<54:bottom_PAC_bit> = result<54:bottom_PAC_bit> EOR PAC<54:bottom_PAC_bit>;
            if HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                if 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);
        else
            if !HaveEnhancedPAC2() then
                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;
                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 HaveFPACCombined() || (HaveFPAC() && !is_combined) then
                    if 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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDAKey_EL1;

    APDAKey_EL1 = APDAKeyHi_EL1<63:0> : APDAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDA else SCTL_EL2.EnDA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnDA;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnDA;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APDBKey_EL1;

    APDBKey_EL1 = APDBKeyHi_EL1<63:0> : APDBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnDB else SCTL_EL2.EnDB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            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 X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIAKey_EL1;

    APIAKey_EL1 = APIAKeyHi_EL1<63:0> : APIAKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTLR_EL1.EnIA else SCTLR_EL2.EnIA;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            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 X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(X, Y, 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)
    boolean TrapEL2;
    boolean TrapEL3;
    bits(1) Enable;
    bits(128) APIBKey_EL1;

    APIBKey_EL1 = APIBKeyHi_EL1<63:0> : APIBKeyLo_EL1<63:0>;
    case PSTATE.EL of
        when EL0
            boolean IsEL1Regime = S1TranslationRegime() == EL1;
            Enable = if IsEL1Regime then SCTL_EL1.EnIB else SCTL_EL2.EnIB;
            TrapEL2 = (EL2Enabled() && HCR_EL2.API == '0' &&
                (HCR_EL2.TGE == '0' || HCR_EL2.E2H == '0'));
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL1
            Enable = SCTL_EL1.EnIB;
            TrapEL2 = EL2Enabled() && HCR_EL2.API == '0';
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL2
            Enable = SCTL_EL2.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = HaveEL(EL3) && SCR_EL3.API == '0';
        when EL3
            Enable = SCTL_EL3.EnIB;
            TrapEL2 = FALSE;
            TrapEL3 = FALSE;

    if Enable == '0' then
        return X;
    elsif TrapEL3 && EL3SDDTrapPriority() then
        UNDEFINED;
    elsif TrapEL2 then
        TrapPACUse(EL2);
    elsif TrapEL3 then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            TrapPACUse(EL3);
    else
        return Auth(X, Y, APIBKey_EL1, FALSE, '1', is_combined);
```

Library pseudocode for aarch64/functions/pac/calcbottompacbit/CalculateBottomPACBit

```
// CalculateBottomPACBit()
// =====

integer CalculateBottomPACBit(bit top_bit)
  integer tsz_field;
  boolean using64k;
  Constraint c;

  if PtrHasUpperAndLowerAddRanges() then
    assert S1TranslationRegime() IN {EL1, EL2};
    if S1TranslationRegime() == EL1 then
      // EL1 translation regime registers
      tsz_field = if top_bit == '1' then UInt(TCR_EL1.T1SZ) else UInt(TCR_EL1.T0SZ);
      using64k = if top_bit == '1' then TCR_EL1.TG1 == '11' else TCR_EL1.TG0 == '01';
    else
      // EL2 translation regime registers
      assert HaveEL(EL2);
      tsz_field = if top_bit == '1' then UInt(TCR_EL2.T1SZ) else UInt(TCR_EL2.T0SZ);
      using64k = if top_bit == '1' then TCR_EL2.TG1 == '11' else TCR_EL2.TG0 == '01';
  else
    tsz_field = if PSTATE.EL == EL2 then UInt(TCR_EL2.T0SZ) else UInt(TCR_EL3.T0SZ);
    using64k = if PSTATE.EL == EL2 then TCR_EL2.TG0 == '01' else TCR_EL3.TG0 == '01';

  max_limit_tsz_field = (if !HaveSmallTranslationTableExt() then 39 else if using64k then 47 else 48);
  if tsz_field > max_limit_tsz_field then
    // TCR_ELx.TySZ is out of range
    c = ConstrainUnpredictable(Unpredictable\_RESTnSZ);
    assert c IN {Constraint\_FORCE, Constraint\_NONE};
    if c == Constraint\_FORCE then tsz_field = max_limit_tsz_field;
  tszmin = if using64k && AArch64.VAMax() == 52 then 12 else 16;
  if tsz_field < tszmin then
    c = ConstrainUnpredictable(Unpredictable\_RESTnSZ);
    assert c IN {Constraint\_FORCE, Constraint\_NONE};
    if c == Constraint\_FORCE then tsz_field = tszmin;
  return (64-tsz_field);
```



```

// ComputePAC()
// =====

bits(64) ComputePAC(bits(64) data, bits(64) modifier, bits(64) key0, bits(64) key1)
bits(64) workingval;
bits(64) runningmod;
bits(64) roundkey;
bits(64) modk0;
constant bits(64) Alpha = 0xC0AC29B7C97C50DD<63:0>;

integer iterations;
if HavePACQARMA3() then
    iterations = 2;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
else
    iterations = 4;
    RC[0] = 0x0000000000000000<63:0>;
    RC[1] = 0x13198A2E03707344<63:0>;
    RC[2] = 0xA4093822299F31D0<63:0>;
    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 HavePACQARMA3() 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 HavePACQARMA3() then
    workingval = PACSub1(workingval);
else
    workingval = PACSub(workingval);
workingval = PACCellShuffle(workingval);
workingval = PACMult(workingval);
workingval = key1 EOR workingval;
workingval = PACCellInvShuffle(workingval);
if HavePACQARMA3() 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 HavePACQARMA3() 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 Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '0101';
        when '0001' Toutput<4*i+3:4*i> = '1110';
        when '0010' Toutput<4*i+3:4*i> = '1101';
        when '0011' Toutput<4*i+3:4*i> = '1000';
        when '0100' Toutput<4*i+3:4*i> = '1010';
        when '0101' Toutput<4*i+3:4*i> = '1011';
        when '0110' Toutput<4*i+3:4*i> = '0001';
        when '0111' Toutput<4*i+3:4*i> = '1001';
        when '1000' Toutput<4*i+3:4*i> = '0010';
        when '1001' Toutput<4*i+3:4*i> = '0110';
        when '1010' Toutput<4*i+3:4*i> = '1111';
        when '1011' Toutput<4*i+3:4*i> = '0000';
        when '1100' Toutput<4*i+3:4*i> = '0100';
        when '1101' Toutput<4*i+3:4*i> = '1100';
        when '1110' Toutput<4*i+3:4*i> = '0111';
        when '1111' Toutput<4*i+3:4*i> = '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> = RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 2);
    t0<3:0> = t0<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t1<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    t1<3:0> = t1<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 2);
    t2<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 2) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 1);
    t2<3:0> = t2<3:0> EOR RotCell(Sinput<4*(i)+3:4*(i)>, 1);
    t3<3:0> = RotCell(Sinput<4*(i+12)+3:4*(i+12)>, 1) EOR RotCell(Sinput<4*(i+8)+3:4*(i+8)>, 2);
    t3<3:0> = t3<3:0> EOR RotCell(Sinput<4*(i+4)+3:4*(i+4)>, 1);
    Soutput<4*i+3:4*i> = t3<3:0>;
    Soutput<4*(i+4)+3:4*(i+4)> = t2<3:0>;
    Soutput<4*(i+8)+3:4*(i+8)> = t1<3:0>;
    Soutput<4*(i+12)+3:4*(i+12)> = 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 Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1011';
        when '0001' Toutput<4*i+3:4*i> = '0110';
        when '0010' Toutput<4*i+3:4*i> = '1000';
        when '0011' Toutput<4*i+3:4*i> = '1111';
        when '0100' Toutput<4*i+3:4*i> = '1100';
        when '0101' Toutput<4*i+3:4*i> = '0000';
        when '0110' Toutput<4*i+3:4*i> = '1001';
        when '0111' Toutput<4*i+3:4*i> = '1110';
        when '1000' Toutput<4*i+3:4*i> = '0011';
        when '1001' Toutput<4*i+3:4*i> = '0111';
        when '1010' Toutput<4*i+3:4*i> = '0100';
        when '1011' Toutput<4*i+3:4*i> = '0101';
        when '1100' Toutput<4*i+3:4*i> = '1101';
        when '1101' Toutput<4*i+3:4*i> = '0010';
        when '1110' Toutput<4*i+3:4*i> = '0001';
        when '1111' Toutput<4*i+3:4*i> = '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 Tinput<4*i+3:4*i> of
        when '0000' Toutput<4*i+3:4*i> = '1010';
        when '0001' Toutput<4*i+3:4*i> = '1101';
        when '0010' Toutput<4*i+3:4*i> = '1110';
        when '0011' Toutput<4*i+3:4*i> = '0110';
        when '0100' Toutput<4*i+3:4*i> = '1111';
        when '0101' Toutput<4*i+3:4*i> = '0111';
        when '0110' Toutput<4*i+3:4*i> = '0011';
        when '0111' Toutput<4*i+3:4*i> = '0101';
        when '1000' Toutput<4*i+3:4*i> = '1001';
        when '1001' Toutput<4*i+3:4*i> = '1000';
        when '1010' Toutput<4*i+3:4*i> = '0000';
        when '1011' Toutput<4*i+3:4*i> = '1100';
        when '1100' Toutput<4*i+3:4*i> = '1011';
        when '1101' Toutput<4*i+3:4*i> = '0001';
        when '1110' Toutput<4*i+3:4*i> = '0010';
        when '1111' Toutput<4*i+3:4*i> = '0100';
return Toutput;
```

Library pseudocode for aarch64/functions/pac/computepac/RC

```
array bits(64) RC[0..4];
```

Library pseudocode for aarch64/functions/pac/computepac/RotCell

```
// RotCell()
// =====

bits(4) RotCell(bits(4) incell, integer amount)
    bits(8) tmp;
    bits(4) outcell;

    // assert amount>3 || amount<1;
    tmp<7:0> = incell<3:0>:incell<3:0>;
    outcell = tmp<7-amount:4-amount>;
    return outcell;
```

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/pac/ConstPACField

```
// ConstPACField()
// =====
// Returns TRUE if bit<55> can be used to determine the size of the PAC field, FALSE otherwise.

boolean ConstPACField()
    return boolean IMPLEMENTATION_DEFINED "Bit 55 determines the size of the PAC field";
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC

```
// HaveEnhancedPAC()
// =====
// Returns TRUE if support for EnhancedPAC is implemented, FALSE otherwise.

boolean HaveEnhancedPAC()
    return ( HavePACExt()
        && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC functionality" );
```

Library pseudocode for aarch64/functions/pac/pac/HaveEnhancedPAC2

```
// HaveEnhancedPAC2()
// =====
// Returns TRUE if support for EnhancedPAC2 is implemented, FALSE otherwise.

boolean HaveEnhancedPAC2()
    return HasArchVersion(ARMv8p6) || ( HasArchVersion(ARMv8p3) && boolean IMPLEMENTATION_DEFINED "Has enhanced PAC2 functionality" );
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPAC

```
// HaveFPAC()
// =====
// Returns TRUE if support for FPAC is implemented, FALSE otherwise.

boolean HaveFPAC()
    return HaveEnhancedPAC2() && boolean IMPLEMENTATION_DEFINED "Has FPAC functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HaveFPACCombined

```
// HaveFPACCombined()
// =====
// Returns TRUE if support for FPACCombined is implemented, FALSE otherwise.

boolean HaveFPACCombined()
    return HaveFPAC\(\) && boolean IMPLEMENTATION_DEFINED "Has FPAC Combined functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACExt

```
// HavePACExt()
// =====
// Returns TRUE if support for the PAC extension is implemented, FALSE otherwise.

boolean HavePACExt()
    return HasArchVersion\(ARMv8p3\);
```

Library pseudocode for aarch64/functions/pac/pac/HavePACIMP

```
// HavePACIMP()
// =====
// Returns TRUE if support for PAC IMP is implemented, FALSE otherwise.

boolean HavePACIMP()
    return HavePACExt\(\) && boolean IMPLEMENTATION_DEFINED "Has PAC IMP functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA3

```
// HavePACQARMA3()
// =====
// Returns TRUE if support for PAC QARMA3 is implemented, FALSE otherwise.

boolean HavePACQARMA3()
    return HavePACExt\(\) && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA3 functionality";
```

Library pseudocode for aarch64/functions/pac/pac/HavePACQARMA5

```
// HavePACQARMA5()
// =====
// Returns TRUE if support for PAC QARMA5 is implemented, FALSE otherwise.

boolean HavePACQARMA5()
    return HavePACExt\(\) && boolean IMPLEMENTATION_DEFINED "Has PAC QARMA5 functionality";
```

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, AccType\_NORMAL);
    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;
    boolean tbi = EffectiveTBI(A, !data, PSTATE.EL) == '1';
    integer bottom_PAC_bit = CalculateBottomPACBit(A<55>);
    extfield = Replicate(A<55>, 64);

    if tbi then
        original_ptr = A<63:56>:extfield<(56-bottom_PAC_bit)-1:0>:A<bottom_PAC_bit-1:0>;
    else
        original_ptr = extfield<(64-bottom_PAC_bit)-1:0>: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);

    bits(64) preferred_exception_return = ThisInstrAddr();
    ExceptionRecord exception;
    vect_offset = 0;
    exception = ExceptionSyndrome(Exception\_PACTrap);
    AArch64.TakeException(target_el, exception, preferred_exception_return, vect_offset);
```

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()
    boolean mask_active;

    route_to_el3 = HaveEL(EL3) && SCR_EL3.EA == '1';
    route_to_el2 = (EL2Enabled() &&
        (HCR_EL2.TGE == '1' || HCR_EL2.AM0 == '1'));

    target = if route_to_el3 then EL3 elsif route_to_el2 then EL2 else EL1;

    if target == EL1 then
        mask_active = PSTATE.EL IN {EL0, EL1};
    elsif HaveVirtHostExt() && target == EL2 && HCR_EL2.<E2H,TGE> == '11' then
        mask_active = PSTATE.EL IN {EL0, EL2};
    else
        mask_active = PSTATE.EL == target;

    mask_set = (PSTATE.A == '1' && (!HaveDoubleFaultExt() || SCR_EL3.EA == '0' ||
        PSTATE.EL != EL3 || SCR_EL3.NMEA == '0'));
    intdis = Halted() || ExternalDebugInterruptsDisabled(target);
    masked = (UInt(target) < UInt(PSTATE.EL)) || intdis || (mask_active && mask_set);

    // Check for a masked Physical SError pending that can be synchronized
    // by an Error synchronization event.
    if masked && IsSynchronizablePhysicalSErrorPending() then
        // This function might be called for an interworking case, and INTdis is masking
        // the SError interrupt.
        if ELUsingAArch32(S1TranslationRegime()) then
            syndrome32 = AArch32.PhysicalSErrorSyndrome();
            DISR = AArch32.ReportDeferredSError(syndrome32.AET, syndrome32.ExT);
        else
            implicit_esb = FALSE;
            syndrome64 = AArch64.PhysicalSErrorSyndrome(implicit_esb);
            DISR_EL1 = AArch64.ReportDeferredSError(syndrome64);
            ClearPendingPhysicalSError(); // Set ISR_EL1.A to 0

    return;
```

Library pseudocode for aarch64/functions/ras/AArch64.PhysicalSErrorSyndrome

```
// Return the SError syndrome
bits(25) AArch64.PhysicalSErrorSyndrome(boolean implicit_esb);
```

Library pseudocode for aarch64/functions/ras/AArch64.ReportDeferredSError

```
// AArch64.ReportDeferredSError()
// =====
// Generate deferred SError syndrome

bits(64) AArch64.ReportDeferredSError(bits(25) syndrome)
    bits(64) target;
    target<31> = '1'; // A
    target<24> = syndrome<24>; // IDS
    target<23:0> = syndrome<23:0>; // ISS
    return target;
```

Library pseudocode for aarch64/functions/ras/AArch64.vESBOperation

```
// AArch64.vESBOperation()
// =====
// Perform the AArch64 ESB operation for virtual SError interrupts, either for ESB
// executed in AArch64 state, or for ESB in AArch32 state with EL2 using AArch64 state

AArch64.vESBOperation()
    assert PSTATE.EL IN {EL0, EL1} && EL2Enabled();

    // If physical SError interrupts are routed to EL2, and TGE is not set, then a virtual
    // SError interrupt might be pending
    vSEI_enabled = HCR_EL2.TGE == '0' && HCR_EL2.AMO == '1';
    vSEI_pending = vSEI_enabled && HCR_EL2.VSE == '1';
    vintdis      = Halted() || ExternalDebugInterruptsDisabled(EL1);
    vmasked      = vintdis || PSTATE.A == '1';

    // Check for a masked virtual SError pending
    if vSEI_pending && vmasked then
        // This function might be called for the interworking case, and INTdis is masking
        // the virtual SError interrupt.
        if ELUsingAArch32(EL1) then
            VDISR = AArch32.ReportDeferredSError(VDFSR<15:14>, VDFSR<12>);
        else
            VDISR_EL2 = AArch64.ReportDeferredSError(VSESR_EL2<24:0>);
            HCR_EL2.VSE = '0'; // Clear pending virtual SError

    return;
```

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();

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetGeneralRegisters

```
// AArch64.ResetGeneralRegisters()
// =====

AArch64.ResetGeneralRegisters()

    for i = 0 to 30
        X[i] = bits(64) UNKNOWN;

    return;
```

Library pseudocode for aarch64/functions/registers/AArch64.ResetSIMDFPRegisters

```
// AArch64.ResetSIMDFPRegisters()
// =====

AArch64.ResetSIMDFPRegisters()

    for i = 0 to 31
        V[i] = 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;
    if HaveEL(EL3) then
        SP_EL3 = bits(64) UNKNOWN;
        SPSR_EL3 = bits(64) UNKNOWN;
        ELR_EL3 = 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(boolean cold_reset);
```

Library pseudocode for aarch64/functions/registers/PC

```
// PC - non-assignment form
// =====
// Read program counter.

bits(64) PC[]
    return _PC;
```

Library pseudocode for aarch64/functions/registers/SP

```
// SP[] - assignment form
// =====
// Write to stack pointer from either a 32-bit or a 64-bit value.

SP[] = bits(width) value
  assert width IN {32,64};
  if PSTATE.SP == '0' then
    SP_EL0 = ZeroExtend(value);
  else
    case PSTATE.EL of
      when EL0 SP_EL0 = ZeroExtend(value);
      when EL1 SP_EL1 = ZeroExtend(value);
      when EL2 SP_EL2 = ZeroExtend(value);
      when EL3 SP_EL3 = ZeroExtend(value);
  return;

// SP[] - non-assignment form
// =====
// Read stack pointer with implicit slice of 8, 16, 32 or 64 bits.

bits(width) SP[]
  assert width IN {8,16,32,64};
  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/V

```
// V[] - assignment form
// =====
// Write to SIMD&FP register with implicit extension from
// 8, 16, 32, 64 or 128 bits.

V[integer n] = bits(width) value
  assert n >= 0 && n <= 31;
  assert width IN {8,16,32,64,128};
  _V[n] = ZeroExtend(value);
  return;

// 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]
  assert n >= 0 && n <= 31;
  assert width IN {8,16,32,64,128};
  return _V[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]
  assert n >= 0 && n <= 31;
  assert part IN {0, 1};
  if part == 0 then
    assert width IN {8,16,32,64};
    return _V[n]<width-1:0>;
  else
    assert width IN {32,64};
    return _V[n]<(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] = bits(width) value
  assert n >= 0 && n <= 31;
  assert part IN {0, 1};
  if part == 0 then
    assert width IN {8,16,32,64};
    _V[n] = ZeroExtend(value);
  else
    assert width == 64;
    _V[n]<(width * 2)-1:width> = value<width-1:0>;
```

Library pseudocode for aarch64/functions/registers/X

```
// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value.

X[integer n] = bits(width) value
    assert n >= 0 && n <= 31;
    assert width IN {32,64};
    if n != 31 then
        _R[n] = ZeroExtend(value);
    return;

// X[] - assignment form
// =====
// Write to general-purpose register from either a 32-bit or a 64-bit value,
// where the size of the value is passed as an argument.

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 from general-purpose register with implicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);

// X[] - non-assignment form
// =====
// Read from general-purpose register with an explicit slice of 8, 16, 32 or 64 bits.

bits(width) X[integer n, integer width]
    assert n >= 0 && n <= 31;
    assert width IN {8,16,32,64};
    if n != 31 then
        return _R[n]<width-1:0>;
    else
        return Zeros(width);
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTL

```
// CNTKCTL[] - non-assignment form
// =====

CNTKCTLType CNTKCTL[]
    bits(64) r;
    if IsInHost() then
        r = CNTKCTL_EL2;
        return r;
    r = CNTKCTL_EL1;
    return r;
```

Library pseudocode for aarch64/functions/sysregisters/CNTKCTLType

```
type CNTKCTLType;
```

Library pseudocode for aarch64/functions/sysregisters/CPACR

```
// CPACR[] - non-assignment form
// =====

CPACRType CPACR[]
  bits(64) r;
  if IsInHost\(\) then
    r = CPTR_EL2;
    return r;
  r = CPACR_EL1;
  return r;
```

Library pseudocode for aarch64/functions/sysregisters/CPACRType

```
type CPACRType;
```

Library pseudocode for aarch64/functions/sysregisters/ELR

```
// ELR[] - non-assignment form
// =====

bits(64) ELR[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[] - non-assignment form
// =====

bits(64) ELR[]
  assert PSTATE.EL != EL0;
  return ELR[PSTATE.EL];

// ELR[] - assignment form
// =====

ELR[bits(2) el] = bits(64) value
  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;

// ELR[] - assignment form
// =====

ELR[] = bits(64) value
  assert PSTATE.EL != EL0;
  ELR[PSTATE.EL] = value;
  return;
```

Library pseudocode for aarch64/functions/sysregisters/ESR

```
// ESR[] - non-assignment form
// =====

ESRType ESR[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[] - non-assignment form
// =====

ESRType ESR[]
  return ESR\[S1TranslationRegime\(\)\];

// ESR[] - assignment form
// =====

ESR[bits(2) regime] = ESRType value
  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;

// ESR[] - assignment form
// =====

ESR[] = ESRType value
  ESR\[S1TranslationRegime\(\)\] = value;
```

Library pseudocode for aarch64/functions/sysregisters/ESRType

```
type ESRType;
```

Library pseudocode for aarch64/functions/sysregisters/FAR

```
// FAR[] - non-assignment form
// =====

bits(64) FAR[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[] - non-assignment form
// =====

bits(64) FAR[]
  return FAR\[S1TranslationRegime\(\)\];

// FAR[] - assignment form
// =====

FAR[bits(2) regime] = bits(64) value
  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;

// FAR[] - assignment form
// =====

FAR[] = bits(64) value
  FAR\[S1TranslationRegime\(\)\] = value;
  return;
```

Library pseudocode for aarch64/functions/sysregisters/MAIR

```
// MAIR[] - non-assignment form
// =====

MAIRType MAIR[bits(2) regime]
  bits(64) r;
  case regime of
    when EL1 r = MAIR_EL1;
    when EL2 r = MAIR_EL2;
    when EL3 r = MAIR_EL3;
    otherwise Unreachable\(\);
  return r;

// MAIR[] - non-assignment form
// =====

MAIRType MAIR[]
  return MAIR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/sysregisters/MAIRType

```
type MAIRType;
```

Library pseudocode for aarch64/functions/sysregisters/SCTLR

```
// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[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;

// SCTLR[] - non-assignment form
// =====

SCTLRType SCTLR[]
    return SCTLR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/sysregisters/SCTLRType

```
type SCTLRType;
```

Library pseudocode for aarch64/functions/sysregisters/VBAR

```
// VBAR[] - non-assignment form
// =====

bits(64) VBAR[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;

// VBAR[] - non-assignment form
// =====

bits(64) VBAR[]
    return VBAR\[S1TranslationRegime\(\)\];
```

Library pseudocode for aarch64/functions/system/AArch64.AllocationTagAccessIsEnabled

```
// AArch64.AllocationTagAccessIsEnabled()
// =====
// Check whether access to Allocation Tags is enabled.

boolean AArch64.AllocationTagAccessIsEnabled(AccType acctype)
    bits(2) el = AArch64.AccessUsesEL(acctype);

    if SCR_EL3.ATA == '0' && el IN {EL0, EL1, EL2} then
        return FALSE;
    elsif HCR_EL2.ATA == '0' && el IN {EL0, EL1} && EL2Enabled() && HCR_EL2.<E2H,TGE> != '11' then
        return FALSE;
    elsif SCTL_EL3.ATA == '0' && el == EL3 then
        return FALSE;
    elsif SCTL_EL2.ATA == '0' && el == EL2 then
        return FALSE;
    elsif SCTL_EL1.ATA == '0' && el == EL1 then
        return FALSE;
    elsif SCTL_EL2.ATA0 == '0' && el == EL0 && EL2Enabled() && HCR_EL2.<E2H,TGE> == '11' then
        return FALSE;
    elsif SCTL_EL1.ATA0 == '0' && el == EL0 && !(EL2Enabled() && HCR_EL2.<E2H,TGE> == '11') then
        return FALSE;
    else
        return TRUE;
```

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.ExecutingBR0rBLR0rRetInstr

```
// AArch64.ExecutingBR0rBLR0rRetInstr()
// =====
// Returns TRUE if current instruction is a BR, BLR, RET, B[L]RA[B][Z], or RETA[B].

boolean AArch64.ExecutingBR0rBLR0rRetInstr()
    if !HaveBTIExt() then return FALSE;

    instr = ThisInstr();
    if instr<31:25> == '1101011' && instr<20:16> == '11111' then
        opc = instr<24:21>;
        return opc != '0101';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/AArch64.ExecutingBTIInstr

```
// AArch64.ExecutingBTIInstr()
// =====
// Returns TRUE if current instruction is a BTI.

boolean AArch64.ExecutingBTIInstr()
    if !HaveBTIExt() 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

```
// 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.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

```
// 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.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.NextRandomTagBit

```
// AArch64.NextRandomTagBit()
// =====
// Generate a random bit suitable for generating a random Allocation Tag.

bit AArch64.NextRandomTagBit()
    bits(16) lfsr = RGSr_EL1.SEED;
    bit top = lfsr<5> EOR lfsr<3> EOR lfsr<2> EOR lfsr<0>;
    RGSr_EL1.SEED = 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

```
// 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

```
// 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

```
// 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

```
// 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

```
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 == EL0 then 35 else 36;
        return SCTLR[<index> == '0';
    else
        return FALSE;
```

Library pseudocode for aarch64/functions/system/BTypeNext

```
bits(2) BTypeNext;
```

Library pseudocode for aarch64/functions/system/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 is described by the Pseudocode used when
// GCR_EL1.RRND is 0, but with a non-deterministic implementation of NextRandomTagBit(). Implementations
// may choose to behave the same as GCR_EL1.RRND=0.
bits(4) ChooseRandomNonExcludedTag(bits(16) exclude_in);
```

Library pseudocode for aarch64/functions/system/InGuardedPage

```
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()
    assert(HaveFeatHCX());
    if HaveEL(EL3) && SCR_EL3.HXEn == '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/instrs/branch/eret/AArch64.ExceptionReturn

```
// AArch64.ExceptionReturn()
// =====

AArch64.ExceptionReturn(bits(64) new_pc_in, bits(64) spsr)
    bits(64) new_pc = new_pc_in;
    if HaveIESB\(\) then
        sync_errors = SCTLR{}.IESB == '1';
        if HaveDoubleFaultExt\(\) then
            sync_errors = sync_errors || (SCR_EL3.<EA,NMEA> == '11' && PSTATE.EL == EL3);
        if sync_errors then
            SynchronizeErrors\(\);
            iesb_req = TRUE;
            TakeUnmaskedPhysicalSErrorInterrupts(iesb_req);
    SynchronizeContext\(\);

    // Attempts to change to an illegal state will invoke the Illegal Execution state mechanism
    bits(2) source_el = PSTATE.EL;
    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);

    if UsingAArch32\(\) then
        // 32 most significant bits are ignored.
        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/instrs/countop/CountOp

```
enumeration CountOp {CountOp_CLZ, CountOp_CLS, CountOp_CNT};
```

Library pseudocode for aarch64/instrs/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/instrs/extendreg/ExtendReg

```
// ExtendReg()
// =====
// Perform a register extension and shift

bits(N) ExtendReg(integer reg, ExtendType exttype, integer shift)
    assert shift >= 0 && shift <= 4;
    bits(N) val = X[reg];
    boolean unsigned;
    integer 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_SXTX unsigned = FALSE; len = 64;
        when ExtendType_UXTB unsigned = TRUE; len = 8;
        when ExtendType_UXTH unsigned = TRUE; len = 16;
        when ExtendType_UXTW unsigned = TRUE; len = 32;
        when ExtendType_UXTX unsigned = TRUE; len = 64;

    // Note the extended width of the intermediate value and
    // that sign extension occurs from bit <len+shift-1>, not
    // from bit <len-1>. This is equivalent to the instruction
    // [SU]BFIZ Rtmp, Rreg, #shift, #len
    // It may also be seen as a sign/zero extend followed by a shift:
    // LSL(Extend(val<len-1:0>, N, unsigned), shift);

    len = Min(len, N - shift);
    return Extend(val<len-1:0> : Zeros(shift), N, unsigned);
```

Library pseudocode for aarch64/instrs/extendreg/ExtendType

```
enumeration ExtendType {ExtendType_SXTB, ExtendType_SXTH, ExtendType_SXTW, ExtendType_SXTX,
                        ExtendType_UXTB, ExtendType_UXTH, ExtendType_UXTW, ExtendType_UXTX};
```

Library pseudocode for aarch64/instrs/float/arithmetic/max-min/fpmaxminop/FPMaxMinOp

```
enumeration FPMaxMinOp {FPMaxMinOp_MAX, FPMaxMinOp_MIN,
                        FPMaxMinOp_MAXNUM, FPMaxMinOp_MINNUM};
```

Library pseudocode for aarch64/instrs/float/arithmatic/unary/fpunaryop/FPUnaryOp

```
enumeration FPUnaryOp    {FPUnaryOp_ABS, FPUnaryOp_MOV,
                          FPUnaryOp_NEG, FPUnaryOp_SQRT};
```

Library pseudocode for aarch64/instrs/float/convert/fpconvop/FPConvOp

```
enumeration FPConvOp    {FPConvOp_CVT_FtoI, FPConvOp_CVT_ItoF,
                          FPConvOp_MOV_FtoI, FPConvOp_MOV_ItoF
                          , FPConvOp_CVT_FtoI_JS
};
```

Library pseudocode for aarch64/instrs/integer/bitfield/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)
    integer S = UInt(imms);
    integer R = UInt(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;
```



```

// 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)
    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]
    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

    // From a software perspective, the remaining code is equivalent to:
    //  esize = 1 << len;
    //  d = UInt(diff<len-1:0>);
    //  welem = ZeroExtend(Ones(S + 1), esize);
    //  telem = ZeroExtend(Ones(d + 1), esize);
    //  wmask = Replicate(ROR(welem, R));
    //  tmask = Replicate(telem);
    //  return (wmask, tmask);

    // 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/instrs/integer/ins-ext/insert/movewide/movewideop/MoveWideOp

```

enumeration MoveWideOp {MoveWideOp_N, MoveWideOp_Z, MoveWideOp_K};

```

Library pseudocode for aarch64/instrs/integer/logical/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)
    integer S = UInt(imms);
    integer R = UInt(immr);
    integer width = if sf == '1' then 64 else 32;

    // element size must equal total immediate size
    if sf == '1' && !((immN:imms) IN {'1xxxxxx'}) then
        return FALSE;
    if sf == '0' && !((immN:imms) IN {'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/instrs/integer/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/instrs/integer/shiftreg/ShiftReg

```
// ShiftReg()
// =====
// Perform shift of a register operand

bits(N) ShiftReg(integer reg, ShiftType shifttype, integer amount)
    bits(N) result = X[reg];
    case shifttype of
        when ShiftType_LSL result = LSL(result, amount);
        when ShiftType_LSR result = LSR(result, amount);
        when ShiftType_ASR result = ASR(result, amount);
        when ShiftType_ROR result = ROR(result, amount);
    return result;
```

Library pseudocode for aarch64/instrs/integer/shiftreg/ShiftType

```
enumeration ShiftType {ShiftType_LSL, ShiftType_LSR, ShiftType_ASR, ShiftType_ROR};
```

Library pseudocode for aarch64/instrs/logicalop/LogicalOp

```
enumeration LogicalOp {LogicalOp_AND, LogicalOp_EOR, LogicalOp_ORR};
```

Library pseudocode for aarch64/instrs/memory/memop/MemAtomicOp

```
enumeration MemAtomicOp {MemAtomicOp_ADD,
    MemAtomicOp_BIC,
    MemAtomicOp_EOR,
    MemAtomicOp_ORR,
    MemAtomicOp_SMAX,
    MemAtomicOp_SMIN,
    MemAtomicOp_UMAX,
    MemAtomicOp_UMIN,
    MemAtomicOp_SWP};
```

Library pseudocode for aarch64/instrs/memory/memop/MemOp

```
enumeration MemOp {MemOp_LOAD, MemOp_STORE, MemOp_PREFETCH};
```

Library pseudocode for aarch64/instrs/memory/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/instrs/system/barriers/barrierop/MemBarrierOp

```
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/instrs/system/hints/syshintop/SystemHintOp

```
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_CSDB
};
```

Library pseudocode for aarch64/instrs/system/register/cpsr/pstatefield/PSTATEField

```
enumeration PSTATEField {PSTATEField_DAIFSet, PSTATEField_DAIFClr,
                         PSTATEField_PAN, // Armv8.1
                         PSTATEField_UA0, // Armv8.2
                         PSTATEField_DIT, // Armv8.4
                         PSTATEField_SSBS,
                         PSTATEField_TC0, // Armv8.5
                         PSTATEField_ALLINT,
                         PSTATEField_SP
                         };
```

Library pseudocode for aarch64/instrs/system/sysops/at/AArch64.AT

```
// AArch64.AT()
// =====
// Perform address translation as per AT instructions.

AArch64.AT(bits(64) address, TranslationStage stage_in, bits(2) el_in, ATAccess ataccess)
    TranslationStage stage = stage_in;
    bits(2) el = el_in;
    // For stage 1 translation, when HCR_EL2.{E2H, TGE} is {1,1} and requested EL is EL1,
    // the EL2&0 translation regime is used.
    if HCR_EL2.<E2H, TGE> == '11' && el == EL1 && stage == TranslationStage_1 then
        el = EL2;
    if HaveEL(EL3) && stage == TranslationStage_12 && !EL2Enabled() then
        stage = TranslationStage_1;

    acctype = if ataccess IN {ATAccess_Read, ATAccess_Write} then AccType_AT else AccType_ATPAN;
    iswrite = ataccess IN {ATAccess_WritePAN, ATAccess_Write};
    aligned = TRUE;
    ispriv = el != EL0;

    fault = NoFault();
    fault.acctype = acctype;
    fault.write = iswrite;

    Regime regime;
    if stage == TranslationStage_12 then
        regime = Regime_EL10;
    else
        regime = TranslationRegime(el, acctype);

    AddressDescriptor addrdesc;
    ss = SecurityStateAtEL(el);
    if (el == EL0 && ELUsingAArch32(EL1)) || (el != EL0 && ELUsingAArch32(el)) then
        if regime == Regime_EL2 || TTBCR.EAE == '1' then
            (fault, addrdesc) = AArch32.S1TranslateLD(fault, regime, ss, address<31:0>, acctype,
                aligned, iswrite, ispriv);
        else
            (fault, addrdesc, -) = AArch32.S1TranslateSD(fault, regime, ss, address<31:0>, acctype,
                aligned, iswrite, ispriv);
    else
        (fault, addrdesc) = AArch64.S1Translate(fault, regime, ss, address, acctype, aligned,
            iswrite, ispriv);

    if stage == TranslationStage_12 && fault.statuscode == Fault_None then
        boolean s2fslwalk;
        boolean slaarch64;
        if ELUsingAArch32(EL1) && regime == Regime_EL10 && EL2Enabled() then
            addrdesc.vaddress = ZeroExtend(address);
            s2fslwalk = FALSE;
            (fault, addrdesc) = AArch32.S2Translate(fault, addrdesc, ss, s2fslwalk, acctype,
                aligned, iswrite, ispriv);
        elsif regime == Regime_EL10 && EL2Enabled() then
            slaarch64 = TRUE;
            s2fslwalk = FALSE;
            (fault, addrdesc) = AArch64.S2Translate(fault, addrdesc, slaarch64, ss, s2fslwalk,
                acctype, aligned, iswrite, ispriv);

    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
        if IsExternalAbort(fault) || (PSTATE.EL == EL1 && fault.s2fslwalk) then
            PAR_EL1 = bits(64) UNKNOWN;
            AArch64.Abort(address, addrdesc.fault);

    AArch64.EncodePAR(regime, addrdesc);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/at/AArch64.EncodePAR

```
// AArch64.EncodePAR()
// =====
// Encode PAR register with result of translation.

AArch64.EncodePAR(Regime regime, AddressDescriptor addrdesc)
    PAR_EL1 = Zeros();
    paspace = addrdesc.paddress.paspace;

    if !IsFault(addrdesc) then
        PAR_EL1.F = '0';
        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));
        PAR_EL1.PA = addrdesc.paddress.address<52-1:12>;
        PAR_EL1.ATTR = ReportedPARAttrs(EncodePARAttrs(addrdesc.memattrs));
        PAR_EL1<10> = bit IMPLEMENTATION_DEFINED "Non-Faulting PAR";
    else
        PAR_EL1.F = '1';
        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/instrs/system/sysops/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;
```



```

// AArch64.DC()
// =====
// Perform Data Cache Operation.

AArch64.DC(bits(64) regval, CacheType cachetype, CacheOp cacheop, CacheOpScope opscope_in)
CacheOpScope opscope = opscope_in;
AccType acctype = AccType\_DC;
CacheRecord cache;

cache.acctype = acctype;
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.set, cache.way, cache.level) = DecodeSW(regval, cachetype);
    if (cacheop == CacheOp\_Invalidate && PSTATE.EL == EL1 && EL2Enabled() &&
        (HCR_EL2.SWI0 == '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\_PoDP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP"
    opscope = CacheOpScope\_PoP;
if opscope == CacheOpScope\_PoP && boolean IMPLEMENTATION_DEFINED "Memory system does not supports PoP"
    opscope = CacheOpScope\_PoC;
need_translate = DCInstNeedsTranslation(opscope);
iswrite = cacheop == CacheOp\_Invalidate;
vaddress = regval;

size = 0; // by default no watchpoint address
if iswrite then
    size = integer IMPLEMENTATION_DEFINED "Data Cache Invalidate Watchpoint Size";
    assert size >= 4*(2^(UInt(CTR_EL0.DminLine))) && size <= 2048;
    assert UInt(size<32:0> AND (size-1)<32:0>) == 0; // size is power of 2
    vaddress = Align(regval, size);

cache.translated = need_translate;
cache.vaddress = vaddress;

if need_translate then
    wasaligned = TRUE;
    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);
    if IsFault(memaddrdesc) then
        AArch64.Abort(regval, memaddrdesc.fault);

    memattrs = memaddrdesc.memattrs;
    cache.paddress = memaddrdesc.paddress;
    cache.cpas = CPASAtPAS(memaddrdesc.paddress.paspace);
    if opscope IN {CacheOpScope\_PoC, CacheOpScope\_PoP, CacheOpScope\_PoDP} then
        cache.shareability = memattrs.shareability;

```

```

    else
        cache.shareability = Shareability\_NSH;
else
    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;

CACHE\_OP(cache);
return;

```

Library pseudocode for aarch64/instrs/system/sysops/dc/AArch64.MemZero

```

// AArch64.MemZero()
// =====

AArch64.MemZero(bits(64) regval, CacheType cachetype)

    AccType acctype = AccType\_DCZVA;
    boolean iswrite = TRUE;
    boolean wasaligned = TRUE;

    integer size = 4*(2^(UInt(DCZID_EL0.BS)));
    bits(64) vaddress = Align(regval, size);

    memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

    if IsFault(memaddrdesc) then
        if IsDebugException(memaddrdesc.fault) then
            AArch64.Abort(vaddress, memaddrdesc.fault);
        else
            AArch64.Abort(regval, memaddrdesc.fault);
    else
        if cachetype == CacheType\_Data then
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);
        elsif cachetype == CacheType\_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
        elsif cachetype == CacheType\_Data\_Tag then
            if HaveMTEExt() then AArch64.TagMemZero(vaddress, size);
            AArch64.DataMemZero(regval, vaddress, memaddrdesc, size);

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;
    AccType acctype = AccType_IC;

    cache.acctype = acctype;
    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 || (opscope == CacheOpScope_ALLU && PSTATE.EL == EL1
            && EL2Enabled() && HCR_EL2.FB == '1')) 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;

        bits(64) vaddress = regval;
        need_translate = ICInstNeedsTranslation(opscope);

        cache.vaddress = regval;
        cache.shareability = Shareability_NSH;
        cache.translated = need_translate;

        if !need_translate then
            cache.address = FullAddress UNKNOWN;
            CACHE_OP(cache);
            return;
        iswrite = FALSE;
        wasaligned = TRUE;
        size = 0;
        memaddrdesc = AArch64.TranslateAddress(vaddress, acctype, iswrite, wasaligned, size);

        if IsFault(memaddrdesc) then
            AArch64.Abort(regval, memaddrdesc.fault);

        cache.cpas = CPASAtPAS(memaddrdesc.address.paspace);
        cache.address = memaddrdesc.address;

```

```
CACHE_OP(cache);  
return;
```

Library pseudocode for aarch64/instrs/system/sysops/predictionrestrict/RestrictPrediction

```
// RestrictPrediction()  
// =====  
// Clear all predictions in the context.  
  
AArch64.RestrictPrediction(bits(64) val, RestrictType restriction)  
  
    ExecutionCntxt c;  
    target_el      = val<25:24>;  
  
    // If the instruction is executed at an EL lower than the specified  
    // level, it is treated as a NOP.  
    if UInt(target_el) > UInt(PSTATE.EL) then return;  
  
    bit ns = val<26>;  
    ss = TargetSecurityState(ns);  
  
    c.security = ss;  
    c.target_el = target_el;  
  
    if EL2Enabled() && !IsInHost() then  
        if PSTATE.EL IN {EL0, EL1} then  
            c.is_vmid_valid = TRUE;  
            c.all_vmid      = FALSE;  
            c.vmid          = VMID[];  
  
            elsif target_el IN {EL0, 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[];  
  
            elsif target_el == EL0 then  
                c.is_asid_valid = TRUE;  
                c.all_asid      = val<16> == '1';  
                c.asid          = val<15:0>;           // Only valid if val<16> == '0';  
            else  
                c.is_asid_valid = FALSE;  
  
        c.restriction = restriction;  
        RESTRICT_PREDICTIONS(c);
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SysOp

```
// 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 '100 0111 1000 000' return Sys_AT; // S1E2R
    when '110 0111 1000 000' return Sys_AT; // S1E3R
    when '000 0111 1000 001' return Sys_AT; // S1E1W
    when '100 0111 1000 001' return Sys_AT; // S1E2W
    when '110 0111 1000 001' return Sys_AT; // S1E3W
    when '000 0111 1000 010' return Sys_AT; // S1E0R
    when '000 0111 1000 011' return Sys_AT; // S1E0W
    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 '011 0111 0100 001' return Sys_DC; // ZVA
    when '000 0111 0110 001' return Sys_DC; // IVAC
    when '000 0111 0110 010' return Sys_DC; // ISW
    when '011 0111 1010 001' return Sys_DC; // CVAC
    when '000 0111 1010 010' return Sys_DC; // CSW
    when '011 0111 1011 001' return Sys_DC; // CVAU
    when '011 0111 1110 001' return Sys_DC; // CIVAC
    when '000 0111 1110 010' return Sys_DC; // CISW
    when '011 0111 1101 001' return Sys_DC; // CVADP
    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 '100 1000 0000 001' return Sys_TLBI; // IPAS2E1IS
    when '100 1000 0000 101' return Sys_TLBI; // IPAS2LE1IS
    when '000 1000 0011 000' return Sys_TLBI; // VMALLE1IS
    when '100 1000 0011 000' return Sys_TLBI; // ALLE2IS
    when '110 1000 0011 000' return Sys_TLBI; // ALLE3IS
    when '000 1000 0011 001' return Sys_TLBI; // VAE1IS
    when '100 1000 0011 001' return Sys_TLBI; // VAE2IS
    when '110 1000 0011 001' return Sys_TLBI; // VAE3IS
    when '000 1000 0011 010' return Sys_TLBI; // ASIDE1IS
    when '000 1000 0011 011' return Sys_TLBI; // VAAE1IS
    when '100 1000 0011 100' return Sys_TLBI; // ALLE1IS
    when '000 1000 0011 101' return Sys_TLBI; // VALE1IS
    when '100 1000 0011 101' return Sys_TLBI; // VALE2IS
    when '110 1000 0011 101' return Sys_TLBI; // VALE3IS
    when '100 1000 0011 110' return Sys_TLBI; // VMALLS12E1IS
    when '000 1000 0011 111' return Sys_TLBI; // VAALE1IS
    when '100 1000 0100 001' return Sys_TLBI; // IPAS2E1
    when '100 1000 0100 101' return Sys_TLBI; // IPAS2LE1
    when '000 1000 0111 000' return Sys_TLBI; // VMALLE1
    when '100 1000 0111 000' return Sys_TLBI; // ALLE2
    when '110 1000 0111 000' return Sys_TLBI; // ALLE3
    when '000 1000 0111 001' return Sys_TLBI; // VAE1
    when '100 1000 0111 001' return Sys_TLBI; // VAE2
    when '110 1000 0111 001' return Sys_TLBI; // VAE3
    when '000 1000 0111 010' return Sys_TLBI; // ASIDE1
    when '000 1000 0111 011' return Sys_TLBI; // VAAE1
    when '100 1000 0111 100' return Sys_TLBI; // ALLE1
    when '000 1000 0111 101' return Sys_TLBI; // VALE1
    when '100 1000 0111 101' return Sys_TLBI; // VALE2
    when '110 1000 0111 101' return Sys_TLBI; // VALE3
    when '100 1000 0111 110' return Sys_TLBI; // VMALLS12E1
    when '000 1000 0111 111' return Sys_TLBI; // VAALE1
  return Sys_SYS;
```

Library pseudocode for aarch64/instrs/system/sysops/sysop/SystemOp

```
enumeration SystemOp {Sys_AT, Sys_DC, Sys_IC, Sys_TLBI, Sys_SYS};
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.DTLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBMemAttr 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 shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBMemAttr 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.level      = TLBILevel\_Any;
    r.attr       = attr;
    r.asid       = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.DTLBI_VA

```
// AArch32.DTLBI_VA()
// =====
// Invalidate by VA all stage 1 data TLB entries in the indicated shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.DTLBI_VA(Shareability security, Regime regime, bits(16) vmid,
                 Shareability shareability, 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.level       = level;
    r.attr        = attr;
    r.asid        = Zeros(8) : Rt<7:0>;
    r.address     = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.

AArch32.ITLBI_ALL(Shareability security, Regime regime, Shareability shareability, 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 shareability != Shareability_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
                  TLBMemAttr 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.level      = TLBIlevel\_Any;
    r.attr       = attr;
    r.asid       = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.ITLBI_VA

```
// AArch32.ITLBI_VA()
// =====
// Invalidate by VA all stage 1 instruction TLB entries in the indicated shareability 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.
// When the indicated level is
//     TLBIlevel_Any : this applies to TLB entries at all levels
//     TLBIlevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.ITLBI_VA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, TLBIlevel level, TLBMemAttr 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.level      = level;
    r.attr       = attr;
    r.asid       = Zeros(8) : Rt<7:0>;
    r.address    = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBMemAttr 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 shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
    TLBMemAttr 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.level      = TLBILevel\_Any;
    r.attr       = attr;
    r.asid       = Zeros(8) : Rt<7:0>;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_IPAS2

```
// AArch32.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, 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.level      = level;
r.attr       = attr;
r.address    = Zeros(24) : Rt<27:0> : Zeros(12);
r.ipaspace   = PAS\_NonSecure;

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VA

```
// AArch32.TLBI_VA()  
// =====  
// Invalidate by VA all stage 1 TLB entries in the indicated shareability 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.  
// When the indicated level is  
//     TLBILevel_Any : this applies to TLB entries at all levels  
//     TLBILevel_Last : this applies to TLB entries at last level only  
// The indicated attr defines the attributes of the memory operations that must be completed in  
// order to deem this operation to be completed.  
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation  
// are required to complete.
```

```
AArch32.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,  
                Shareability shareability, 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.level      = level;  
r.attr       = attr;  
r.asid       = Zeros(8) : Rt<7:0>;  
r.address    = Zeros(32) : Rt<31:12> : Zeros(12);
```

```
TLBI(r);  
if shareability != Shareability\_NSH then Broadcast(shareability, r);  
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch32.TLBI_VAA

```
// AArch32.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, 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.level        = level;
    r.attr         = attr;
    r.address      = Zeros(32) : Rt<31:12> : Zeros(12);

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability
// 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.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, 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.attr         = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch32.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, 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.attr       = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Invalidation applies to all applicable stage 1 and stage 2 entries.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ALL(SecurityState security, Regime regime, Shareability shareability, TLBIMemAttr attr)
    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 shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability domain.
// Note: stage 1 and stage 2 combined entries are in the scope of this operation.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_ASID(SecurityState security, Regime regime, bits(16) vmid, Shareability shareability,
TLBMemAttr 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.level = TLBILevel\_Any;
r.attr = attr;
r.asid = Xt<63:48>;

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_IPAS2

```
// AArch64.TLBI_IPAS2()
// =====
// Invalidate by IPA all stage 2 only TLB entries in the indicated shareability
// 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.
// When the indicated level is
//     TLBIlevel_Any : this applies to TLB entries at all levels
//     TLBIlevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_IPAS2(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, 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.level    = level;
r.attr     = attr;
r.address  = ZeroExtend(Xt<39:0> : Zeros(12));

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;

TLBI(r);
if shareability != Shareability\_NSH then Broadcast(shareability, r);
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RIPAS2

```
// AArch64.TLBI_RIPAS2()
// =====
// Range invalidate by IPA all stage 2 only TLB entries in the indicated
// shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RIPAS2(SecurityState security, Regime regime, bits(16) vmid,
                   Shareability shareability, 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.level      = level;
    r.attr       = attr;

    bits(2) tg      = Xt<47:46>;
    integer scale   = UInt(Xt<45:44>);
    integer num     = UInt(Xt<43:39>);
    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;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVA

```
// AArch64.TLBI_RVA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, 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.level      = level;
    r.attr       = attr;
    r.asid       = Xt<63:48>;

    boolean valid;

    (valid, r.tg, r.address, r.end_address) = TLBIRange(regime, Xt);

    if !valid then return;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_RVAA

```
// AArch64.TLBI_RVAA()
// =====
// Range invalidate by VA range all stage 1 TLB entries in the indicated
// shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_RVAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, 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.level      = level;
    r.attr       = attr;

    bits(2) tg      = Xt<47:46>;
    integer scale   = UInt(Xt<45:44>);
    integer num     = UInt(Xt<43:39>);
    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 shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VA

```
// AArch64.TLBI_VA()  
// =====  
// Invalidate by VA all stage 1 TLB entries in the indicated shareability 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.  
// When the indicated level is  
//     TLBILevel_Any : this applies to TLB entries at all levels  
//     TLBILevel_Last : this applies to TLB entries at last level only  
// The indicated attr defines the attributes of the memory operations that must be completed in  
// order to deem this operation to be completed.  
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation  
// are required to complete.
```

```
AArch64.TLBI_VA(SecurityState security, Regime regime, bits(16) vmid,  
                Shareability shareability, 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.level       = level;  
r.attr        = attr;  
r.asid        = Xt<63:48>;  
r.address     = ZeroExtend(Xt<43:0> : Zeros(12));  
  
TLBI(r);  
if shareability != Shareability\_NSH then Broadcast(shareability, r);  
return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/AArch64.TLBI_VAA

```
// AArch64.TLBI_VAA()
// =====
// Invalidate by VA all stage 1 TLB entries in the indicated shareability 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.
// When the indicated level is
//     TLBILevel_Any : this applies to TLB entries at all levels
//     TLBILevel_Last : this applies to TLB entries at last level only
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VAA(SecurityState security, Regime regime, bits(16) vmid,
                 Shareability shareability, 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.level      = level;
    r.attr       = attr;
    r.address    = ZeroExtend(Xt<43:0> : Zeros(12));

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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 shareability
// 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.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALL(SecurityState security, Regime regime, bits(16) vmid,
                  Shareability shareability, TLBIMemAttr attr)
    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.attr       = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/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
// shareability domain that match the indicated VMID.
// The indicated attr defines the attributes of the memory operations that must be completed in
// order to deem this operation to be completed.
// When attr is TLBI_ExcludeXS, only operations with XS=0 within the scope of this TLB operation
// are required to complete.

AArch64.TLBI_VMALLS12(SecurityState security, Regime regime, bits(16) vmid,
                     Shareability shareability, TLBIMemAttr attr)
    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.attr        = attr;

    TLBI(r);
    if shareability != Shareability\_NSH then Broadcast(shareability, r);
    return;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/ASID_NONE

```
constant bits(16) ASID_NONE = Zeros();
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/Broadcast

```
// Broadcast()
// =====
// IMPLEMENTATION DEFINED function to broadcast TLBI operation within the indicated shareability
// domain.

Broadcast(Shareability shareability, TLBIRecord r)
    IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/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/instrs/system/sysops/tlbi/HasLargeAddress

```
// HasLargeAddress()
// =====
// Returns TRUE if the regime is configured for 52 bit addresses, FALSE otherwise.

boolean HasLargeAddress(Regime regime)
  if !Have52BitIPAAndPASpaceExt() then
    return FALSE;
  case regime of
    when Regime_EL3
      return TCR_EL3<32> == '1';
    when Regime_EL2
      return TCR_EL2<32> == '1';
    when Regime_EL20
      return TCR_EL2<59> == '1';
    when Regime_EL10
      return TCR_EL1<59> == '1';
    otherwise
      Unreachable();
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBI

```
// TLBI()
// =====
// Performs TLB maintenance of operation on TLB to invalidate the matching transition table entries.

TLBI(TLBIRecord r)
  IMPLEMENTATION_DEFINED;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBILevel

```
enumeration TLBILevel {
  TLBILevel_Any,
  TLBILevel_Last
};
```



```

// TLBIMatch()
// =====
// Determine whether the TLB entry lies within the scope of invalidation

boolean TLBIMatch(TLBIRecord tlbi, TLBRecord entry)
    boolean match;
    case tlbi.op of
        when TLBIOp_DALL, TLBIOp_IALL
            match = (tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime);
        when TLBIOp_DASID, TLBIOp_IASID
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (UseASID(entry.context) && entry.context.nG == '1' &&
                     tlbi.asid == entry.context.asid));
        when TLBIOp_DVA, TLBIOp_IVA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                     entry.context.nG == '0') &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_ALL
            relax_regime = (tlbi.from_aarch64 &&
                            tlbi.regime IN {Regime_EL20, Regime_EL2} &&
                            entry.context.regime IN {Regime_EL20, Regime_EL2});
            match = (tlbi.security == entry.context.ss &&
                    (tlbi.regime == entry.context.regime || relax_regime));
        when TLBIOp_ASID
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (UseASID(entry.context) && entry.context.nG == '1' &&
                     tlbi.asid == entry.context.asid));
        when TLBIOp_IPAS2
            match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    tlbi.ipaspace == entry.context.ipaspace &&
                    tlbi.address<51:entry.blocksize> == entry.context.ia<51:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VAA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VA
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
                    (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
                     entry.context.nG == '0') &&
                    tlbi.address<55:entry.blocksize> == entry.context.ia<55:entry.blocksize> &&
                    (tlbi.level == TLBILevel_Any || !entry.walkstate.istable));
        when TLBIOp_VMALL
            match = (entry.context.includes_s1 &&
                    tlbi.security == entry.context.ss &&
                    tlbi.regime == entry.context.regime &&
                    (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
        when TLBIOp_VMALLS12
            match = (tlbi.security == entry.context.ss &&

```

```

        tlbi.regime == entry.context.regime &&
        (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid));
when TLBIOp_RIPAS2
    match = (!entry.context.includes_s1 && entry.context.includes_s2 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            tlbi.ipaspace == entry.context.ipaspace &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVAA
    match = (entry.context.includes_s1 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));
when TLBIOp_RVA
    match = (entry.context.includes_s1 &&
            tlbi.security == entry.context.ss &&
            tlbi.regime == entry.context.regime &&
            (!UseVMID(entry.context) || tlbi.vmid == entry.context.vmid) &&
            (!UseASID(entry.context) || tlbi.asid == entry.context.asid ||
             entry.context.nG == '0') &&
            (tlbi.tg != '00' && DecodeTLBITG(tlbi.tg) == entry.context.tg) &&
            UInt(tlbi.address) <= UInt(entry.context.ia) &&
            UInt(tlbi.end_address) > UInt(entry.context.ia));

if tlbi.attr == TLBI_ExcludeXS && entry.context.xs == '1' then
    match = FALSE;

return match;

```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIMemAttr

```

enumeration TLBIMemAttr {
    TLBI_AllAttr,
    TLBI_ExcludeXS
};

```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/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_VAA,
    TLBIOp_VA,
    TLBIOp_VMALL,
    TLBIOp_VMALLS12,
    TLBIOp_RIPAS2,
    TLBIOp_RVAA,
    TLBIOp_RVA,
};

```

Library pseudocode for aarch64/instrs/system/sysops/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)
    boolean valid = TRUE;
    bits(64) start = Zeros(64);
    bits(64) end   = Zeros(64);

    bits(2) tg      = Xt<47:46>;
    integer scale   = UInt(Xt<45:44>);
    integer num     = UInt(Xt<43:39>);
    integer tg_bits;

    if tg == '00' then
        return (FALSE, tg, start, end);

    case tg of
        when '01' // 4KB
            tg_bits = 12;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<48:12> = Xt<36:0>;
                start<63:49> = Replicate(Xt<36>, 15);
        when '10' // 16KB
            tg_bits = 14;
            if HasLargeAddress(regime) then
                start<52:16> = Xt<36:0>;
                start<63:53> = Replicate(Xt<36>, 11);
            else
                start<50:14> = Xt<36:0>;
                start<63:51> = Replicate(Xt<36>, 13);
        when '11' // 64KB
            tg_bits = 16;
            start<52:16> = Xt<36:0>;
            start<63:53> = Replicate(Xt<36>, 11);
        otherwise
            Unreachable();

    integer range = (num+1) << (5*scale + 1 + tg_bits);
    end   = start + range<63:0>;

    if end<52> != start<52> then
        // overflow, saturate it
        end = Replicate(start<52>, 64-52) : Ones(52);

    return (valid, tg, start, end);
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/TLBIRecord

```
type TLBIRecord is (
    TLBIOp          op,
    boolean         from_aarch64, // originated as an AArch64 operation
    SecurityState  security,
    Regime         regime,
    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
    bits(2)       tg,           // for range operations, translation granule
)
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID

```
// VMID[]
// =====
// Effective VMID.

bits(16) VMID[]
  if EL2Enabled() then
    if !ELUsingAArch32(EL2) then
      if Have16bitVMID() && 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) && HaveSecureEL2Ext() then
    return Zeros(16);
  else
    return VMID_NONE;
```

Library pseudocode for aarch64/instrs/system/sysops/tlbi/VMID_NONE

```
constant bits(16) VMID_NONE = Zeros();
```

Library pseudocode for aarch64/instrs/vector/arithmetic/binary/uniform/logical/bsl-eor/vbitop/VBitOp

```
enumeration VBitOp {VBitOp_VBIF, VBitOp_VBIT, VBitOp_VBSL, VBitOp_VEOR};
```

Library pseudocode for aarch64/instrs/vector/arithmetic/unary/cmp/compareop/CompareOp

```
enumeration CompareOp {CompareOp_GT, CompareOp_GE, CompareOp_EQ,
                      CompareOp_LE, CompareOp_LT};
```

Library pseudocode for aarch64/instrs/vector/logical/immediateop/ImmediateOp

```
enumeration ImmediateOp {ImmediateOp_MOVI, ImmediateOp_MVNI,
                        ImmediateOp_ORR, ImmediateOp_BIC};
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/Reduce

```
// Reduce()
// =====

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize)
    boolean altfp = HaveAltFP() && !UsingAArch32() && FPCR.AH == '1';
    return Reduce(op, input, esize, altfp);

// Reduce()
// =====
// Perform the operation 'op' on pairs of elements from the input vector,
// reducing the vector to a scalar result. The 'altfp' argument controls
// alternative floating-point behaviour.

bits(esize) Reduce(ReduceOp op, bits(N) input, integer esize, boolean altfp)
    integer half;
    bits(esize) hi;
    bits(esize) lo;
    bits(esize) result;

    if N == esize then
        return input<esize-1:0>;

    half = N DIV 2;
    hi = Reduce(op, input<N-1:half>, esize, altfp);
    lo = Reduce(op, input<half-1:0>, esize, altfp);

    case op of
        when ReduceOp_FMIMUM
            result = FPMinum(lo, hi, FPCR[]);
        when ReduceOp_FMAXNUM
            result = FPMaxNum(lo, hi, FPCR[]);
        when ReduceOp_FMIN
            result = FPMin(lo, hi, FPCR[], altfp);
        when ReduceOp_FMAX
            result = FPMax(lo, hi, FPCR[], altfp);
        when ReduceOp_FADD
            result = FPAdd(lo, hi, FPCR[]);
        when ReduceOp_ADD
            result = lo + hi;

    return result;
```

Library pseudocode for aarch64/instrs/vector/reduce/reduceop/ReduceOp

```
enumeration ReduceOp {ReduceOp_FMIMUM, ReduceOp_FMAXNUM,
    ReduceOp_FMIN, ReduceOp_FMAX,
    ReduceOp_FADD, ReduceOp_ADD};
```

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(bits(64) vaddress, AccType acctype_in, integer size)
  assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
  assert (UsingAArch32\(\) && size IN {2,4}) || size == 4;
  AccType acctype = acctype_in;

  match = FALSE;

  for i = 0 to NumBreakpointsImplemented\(\) - 1
    match_i = AArch64.BreakpointMatch(i, vaddress, acctype, size);
    match = match || match_i;

  if match && HaltOnBreakpointOrWatchpoint\(\) then
    reason = DebugHalt\_Breakpoint;
    Halt(reason);
  elseif match then
    acctype = AccType\_IFETCH;
    iswrite = FALSE;
    return AArch64.DebugFault(acctype, iswrite);
  else
    return NoFault();
```

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, AccType acctype, boolean iswrite, integer size)

  FaultRecord fault = NoFault();
  boolean generate_exception;

  boolean d_side = (IsDataAccess(acctype) || acctype == AccType\_DC);
  boolean i_side = (acctype == AccType\_IFETCH);
  if HaveNV2Ext\(\) && acctype == AccType\_NV2REGISTER then
    mask = '0';
    ss = CurrentSecurityState();
    generate_exception = AArch64.GenerateDebugExceptionsFrom(EL2, ss, mask) && MDCR\_EL1.MDE == '1';
  else
    generate_exception = AArch64.GenerateDebugExceptions() && MDCR\_EL1.MDE == '1';
  halt = HaltOnBreakpointOrWatchpoint();

  if generate_exception || halt then
    if d_side then
      fault = AArch64.CheckWatchpoint(vaddress, acctype, iswrite, size);
    elseif i_side then
      fault = AArch64.CheckBreakpoint(vaddress, acctype, size);

  return fault;
```

Library pseudocode for aarch64/translation/debug/AArch64.CheckWatchpoint

```
// 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(bits(64) vaddress, AccType acctype,
                                     boolean iswrite_in, integer size)
    assert !ELUsingAArch32\(S1TranslationRegime\(\)\);
    boolean iswrite = iswrite_in;

    if acctype == AccType\_DC then
        if !iswrite then
            return NoFault\(\);
    elseif !IsDataAccess(acctype) then
        return NoFault\(\);

    match = FALSE;
    match_on_read = FALSE;
    ispriv = AArch64.AccessUsesEL(acctype) != EL0;

    for i = 0 to NumWatchpointsImplemented\(\) - 1
        if AArch64.WatchpointMatch(i, vaddress, size, ispriv, acctype, iswrite) then
            match = TRUE;
            if DBGWCR\_EL1[i].LSC<0> == '1' then
                match_on_read = TRUE;

    if match && IsAtomicRW(acctype) then
        iswrite = !match_on_read;

    if match && HaltOnBreakpointOrWatchpoint\(\) then
        reason = DebugHalt\_Watchpoint;
        EDWAR = vaddress;
        Halt(reason);
    elseif match then
        return AArch64.DebugFault(acctype, iswrite);
    else
        return NoFault\(\);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.BlockBase

```
// AArch64.BlockBase()
// =====
// Extract the address embedded in a block descriptor pointing to the base of
// a memory block

bits(52) AArch64.BlockBase(bits(64) descriptor, bit ds, TGx tgx, integer level)
    bits(52) blockbase = Zeros();

    if tgx == TGx_4KB && level == 2 then
        blockbase<47:21> = descriptor<47:21>;
    elsif tgx == TGx_4KB && level == 1 then
        blockbase<47:30> = descriptor<47:30>;
    elsif tgx == TGx_4KB && level == 0 then
        blockbase<47:39> = descriptor<47:39>;
    elsif tgx == TGx_16KB && level == 2 then
        blockbase<47:25> = descriptor<47:25>;
    elsif tgx == TGx_16KB && level == 1 then
        blockbase<47:36> = descriptor<47:36>;
    elsif tgx == TGx_64KB && level == 2 then
        blockbase<47:29> = descriptor<47:29>;
    elsif tgx == TGx_64KB && level == 1 then
        blockbase<47:42> = descriptor<47:42>;
    else
        Unreachable();

    if Have52BitPAExt() && tgx == TGx_64KB then
        blockbase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        blockbase<51:48> = descriptor<9:8>;descriptor<49:48>;

    return blockbase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.IASize

```
// AArch64.IASize()
// =====
// Retrieve the number of bits containing the input address

integer AArch64.IASize(bits(6) txsz)
    return 64 - UInt(txsz);
```

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(52) AArch64.NextTableBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) tablebase = Zeros();

    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>;

    if Have52BitPAExt() && tgx == TGx_64KB 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.PageBase

```
// AArch64.PageBase()
// =====
// Extract the address embedded in a page descriptor pointing to the base of
// a memory page

bits(52) AArch64.PageBase(bits(64) descriptor, bit ds, TGx tgx)
    bits(52) pagebase = Zeros();

    case tgx of
        when TGx_4KB    pagebase<47:12> = descriptor<47:12>;
        when TGx_16KB   pagebase<47:14> = descriptor<47:14>;
        when TGx_64KB   pagebase<47:16> = descriptor<47:16>;

    if Have52BitPAExt() && tgx == TGx_64KB then
        pagebase<51:48> = descriptor<15:12>;
    elsif ds == '1' then
        pagebase<51:48> = descriptor<9:8>:descriptor<49:48>;

    return pagebase;
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.PhysicalAddressSize

```
// AArch64.PhysicalAddressSize()
// =====
// Retrieve the number of bits bounding the physical address

integer AArch64.PhysicalAddressSize(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;
        otherwise
            ps = integer IMPLEMENTATION_DEFINED "Reserved Intermediate Physical Address size value";

    if tgx != TGx_64KB && !Have52BitIPAAndPASpaceExt() then
        max_ps = Min(48, AArch64.PAMax());
    else
        max_ps = AArch64.PAMax();

    return Min(ps, max_ps);
```

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);
    stride = granulebits - 3;

    return FINAL_LEVEL - (((iasize-1) - granulebits) DIV stride);
```

Library pseudocode for aarch64/translation/vmsa_addrcalc/AArch64.S2SLTTEnterAddress

```
// AArch64.S2SLTTEnterAddress()
// =====
// Compute the first stage 2 translation table descriptor address within the
// table pointed to by the base at the start level

FullAddress AArch64.S2SLTTEnterAddress(S2TTWParams walkparams, bits(52) ipa,
                                       FullAddress tablebase)

    startlevel = AArch64.S2StartLevel(walkparams);
    iasize     = AArch64.IASize(walkparams.txsz);
    granulebits = TGxGranuleBits(walkparams.tgx);
    stride     = granulebits - 3;
    levels     = FINAL\_LEVEL - startlevel;

    bits(52) index;
    lsb  = levels*stride + granulebits;
    msb  = iasize - 1;
    index = ZeroExtend(ipa<msb:lsb>:Zeros(3));

    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)
    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.TTBaseAddress

```
// AArch64.TTBaseAddress()
// =====
// Retrieve the PA/IPA pointing to the base of the initial translation table

bits(52) AArch64.TTBaseAddress(bits(64) ttb, bits(6) txsz, bits(3) ps,
                               bit ds, TGx tgx, integer startlevel)
    bits(52) tablebase = Zeros();

// Input Address size
iasize      = AArch64.IASize(txsz);
granulebits = TGxGranuleBits(tgx);
stride      = granulebits - 3;
levels      = FINAL_LEVEL - startlevel;

// Base address is aligned to size of the initial translation table in bytes
tsize = (iasize - (levels*stride + granulebits)) + 3;

if (Have52BitPAExt() && tgx == TGx_64KB && ps == '110') || (ds == '1') then
    tsize = Max(tsize, 6);
    tablebase<51:6> = ttb<5:2>:ttb<47:6>;
else
    tablebase<47:1> = ttb<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, TGx tgx, bits(6) txsz,
                                   bits(64) ia, FullAddress tablebase)

// Input Address size
iasize      = AArch64.IASize(txsz);
granulebits = TGxGranuleBits(tgx);
stride      = granulebits - 3;
levels      = FINAL_LEVEL - level;

bits(52) index;
lsb  = levels*stride + granulebits;
msb  = Min(iasize - 1, (lsb + stride) - 1);
index = ZeroExtend(ia<msb:lsb>:Zeros(3));

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.

integer AArch64.AddrTop(bit tbid, AccType acctype, bit tbi)
    if tbid == '1' && acctype == AccType_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(bits(6) txsz, TGx tgx, integer level)
    // Input Address size
    iasize = AArch64.IASize(txsz);
    // Translation size
    tsize = TranslationSize(tgx, level) + ContiguousSize(tgx, level);

    fault = boolean IMPLEMENTATION_DEFINED "Translation fault on misprogrammed contiguous bit";

    return tsize > iasize && fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.DebugFault

```
// AArch64.DebugFault()
// =====
// Return a fault record indicating a hardware watchpoint/breakpoint

FaultRecord AArch64.DebugFault(AccType acctype, boolean iswrite)
    FaultRecord fault;

    fault.statuscode = Fault_Debug;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.ExclusiveFault

```
// AArch64.ExclusiveFault()
// =====

FaultRecord AArch64.ExclusiveFault(AccType acctype, boolean iswrite,
                                   boolean secondstage, boolean s2fslwalk)
    FaultRecord fault;

    fault.statuscode = Fault_Exclusive;
    fault.acctype = acctype;
    fault.write = iswrite;
    fault.secondstage = secondstage;
    fault.s2fslwalk = s2fslwalk;

    return fault;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.IPAIsOutOfRange

```
// AArch64.IPAIsOutOfRange()
// =====
// Check bits not resolved by translation are ZERO

boolean AArch64.IPAIsOutOfRange(bits(52) ipa, S2TTWParams walkparams)
    //Input Address size
    iasize = AArch64.IASize(walkparams.txsz);

    if iasize < 52 then
        return !IsZero(ipa<51: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(TTWState walkstate, bits(3) ps, TGx tgx, bits(64) ia)
    // Output Address size
    oasize = AArch64.PhysicalAddressSize(ps, tgx);

    if oasize < 52 then
        if walkstate.istable then
            baseaddress = walkstate.baseaddress.address;
            return !IsZero(baseaddress<51:oasize>);
        else
            // Output address
            oa = Stage0A(ia, tgx, walkstate);
            return !IsZero(oa.address<51:oasize>);
    else
        return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1HasAlignmentFault

```
// AArch64.S1HasAlignmentFault()
// =====
// Returns whether stage 1 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S1HasAlignmentFault(AccType acctype, boolean aligned,
                                     bit ntlsmid, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    if acctype == AccType_A32LSMD && ntlsmid == '0' && memattrs.device != DeviceType_GRE then
        return TRUE;

    return !aligned || acctype == AccType_DCZVA;
```



```

// AArch64.S1HasPermissionsFault()
// =====
// Returns whether stage 1 access violates permissions of target memory

boolean AArch64.S1HasPermissionsFault(Regime regime, SecurityState ss, TTWState walkstate,
                                       S1TTWParams walkparams, boolean ispriv, AccType acctype,
                                       boolean iswrite)

bit r;
bit w;
bit x;
permissions = walkstate.permissions;

if HasUnprivileged(regime) then
    bit pr;
    bit pw;
    bit ur;
    bit uw;
    // 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);

    pan_access = !(acctype IN {AccType_DC, AccType_IFETCH, AccType_AT, AccType_NV2REGISTER});
    if HavePANExt() && pan_access && !(regime == Regime_EL10 && walkparams.nv1 == '1') then
        bit pan;
        if (boolean IMPLEMENTATION_DEFINED "SCR_EL3.SIF affects EPAN" &&
            CurrentSecurityState() == SS_Secure &&
            walkstate.baseaddress.paspace == PAS_NonSecure &&
            walkparams.sif == '1') 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);

    (r,w,x) = if ispriv then (pr,pw,px) else (ur,uw,ux);
else
    // Apply leaf permissions
    case permissions.ap<2> of
        when '0' (r,w) = ('1','1'); // No effect
        when '1' (r,w) = ('1','0'); // Read-only

    // Apply hierarchical permissions
    case permissions.ap_table<1> of
        when '0' (r,w) = ( r , w ); // No effect
        when '1' (r,w) = ( r , '0'); // Read-only

    x = NOT(permissions.xn OR permissions.xn_table);

    // Prevent execution from writable locations if WXN is set
    x = x AND NOT(walkparams.wxn AND w);
    // Prevent execution from Non-secure space by PE in secure state if SIF is set
    if ss == SS_Secure && walkstate.baseaddress.paspace == PAS_NonSecure then
        x = x AND NOT(walkparams.sif);

    if acctype == AccType_IFETCH then
        if (ConstrainUnpredictable(Unpredictable_INSTRDEVICE) == Constraint_FAULT &&

```

```

        walkstate.memattrs.memtype == MemType_Device) then
    return TRUE;

    return x == '0';
elseif acctype == AccType_DC then
    if iswrite then
        return w == '0';
    else
        // DC from privileged context which do no write cannot Permission fault
        return !ispriv && (r == '0' ||
            (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elseif acctype == AccType_IC then
    // IC instructions do not write
    assert !iswrite;
    impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on ELO IC_IVAU execution";

    // IC from privileged context cannot Permission fault
    return !ispriv && ((r == '0' && impdef_ic_fault) ||
        (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
elseif iswrite then
    return w == '0';
else
    return r == '0';

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S1InvalidTxSZ

```

// AArch64.S1InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 1 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S1InvalidTxSZ(S1TTWParams walkparams)
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasAlignmentFault

```

// AArch64.S2HasAlignmentFault()
// =====
// Returns whether stage 2 output fails alignment requirement on data accesses
// to Device memory

boolean AArch64.S2HasAlignmentFault(AccType acctype, boolean aligned, MemoryAttributes memattrs)
    if acctype == AccType_IFETCH || memattrs.memtype != MemType_Device then
        return FALSE;

    return !aligned || acctype == AccType_DCZVA;

```

Library pseudocode for aarch64/translation/vmsa_faults/AArch64.S2HasPermissionsFault

```
// AArch64.S2HasPermissionsFault()
// =====
// Returns whether stage 2 access violates permissions of target memory

boolean AArch64.S2HasPermissionsFault(boolean s2fslwalk, TTWState walkstate, SecurityState ss,
                                       S2TTWParams walkparams, boolean ispriv, AccType acctype,
                                       boolean iswrite)

    permissions = walkstate.permissions;
    memtype = walkstate.memattrs.memtype;

    r = permissions.s2ap<0>;
    w = permissions.s2ap<1>;

    bit px;
    bit 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 ispriv then px else ux;

    if s2fslwalk && walkparams.ptw == '1' && memtype == MemType_Device then
        return TRUE;
    elsif acctype == AccType_IFETCH then
        constraint = ConstrainUnpredictable(Unpredictable_INSTRDEVICE);
        if constraint == Constraint_FAULT && memtype == MemType_Device then
            return TRUE;
        return x == '0';
    elsif acctype == AccType_DC then
        // AArch32 DC maintenance instructions operating by VA cannot fault.
        if iswrite then
            return !ELUsingAArch32(EL1) && w == '0';
        else
            return ((!ispriv && !ELUsingAArch32(EL1) && r == '0') ||
                    (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif acctype == AccType_IC then
        // IC instructions do not write
        assert !iswrite;
        impdef_ic_fault = boolean IMPLEMENTATION_DEFINED "Permission fault on EL0 IC_IVAU execution";

        return ((!ispriv && !ELUsingAArch32(EL1) && r == '0' && impdef_ic_fault) ||
                (IsCMOWControlledInstruction() && walkparams.cmow == '1' && w == '0'));
    elsif iswrite then
        return w == '0';
    else
        return r == '0';
```

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);
    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       = AArch64.IASize(walkparams.txsz);

    return iasize < sl_min_iasize || iasize > sl_max_iasize;
```

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 '010' return AArch64.PAMax() < 44;
                when '011' return !HaveSmallTranslationTableExt();
                otherwise return FALSE;
        when TGx_16KB
            case walkparams.sl0 of
                when '11' return walkparams.ds == '0';
                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.S2InvalidTxSZ

```
// AArch64.S2InvalidTxSZ()
// =====
// Detect erroneous configuration of stage 2 TxSZ field if the implementation
// does not constrain the value of TxSZ

boolean AArch64.S2InvalidTxSZ(S2TTWParams walkparams, boolean slaarch64)
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);

    return UInt(walkparams.txsz) < mintxsz || UInt(walkparams.txsz) > maxtxsz;
```

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, AccType acctype, Regime regime, S1TTWParams walkparams)
  addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
  // Input Address size
  iasize = AArch64.IASize(walkparams.txsz);

  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>);
```

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.IsS2ResultTagged

```
// AArch64.IsS2ResultTagged()
// =====
// Determine whether the combined output memory attributes of stage 1 and
// stage 2 indicate tagged memory

boolean AArch64.IsS2ResultTagged(MemoryAttributes s2_memattrs, boolean s1_tagged)
  return (
    s1_tagged &&
    (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)
  );
```

Library pseudocode for aarch64/translation/vmsa_memattr/AArch64.S2ApplyFWBMemAttrs

```
// AArch64.S2ApplyFWBMemAttrs()
// =====
// Apply stage 2 forced Write-Back on stage 1 memory attributes.

MemoryAttributes AArch64.S2ApplyFWBMemAttrs(MemoryAttributes s1_memattrs,
                                             bits(4) s2_attr, bits(2) s2_sh)
    MemoryAttributes memattrs;

    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;

    elsif s2_attr<1:0> == '11' then // S2 attr = S1 attr
        memattrs = s1_memattrs;

    elsif s2_attr<1:0> == '10' then // Force writeback
        memattrs.memtype = MemType_Normal;
        memattrs.inner.attrs = MemAttr_WB;
        memattrs.outer.attrs = MemAttr_WB;

        if (s1_memattrs.memtype == MemType_Normal &&
            s1_memattrs.inner.attrs != MemAttr_NC) then
            memattrs.inner.hints = s1_memattrs.inner.hints;
            memattrs.inner.transient = s1_memattrs.inner.transient;
        else
            memattrs.inner.hints = MemHint_RWA;
            memattrs.inner.transient = FALSE;

        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;

    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;

    s2_shareability = DecodeShareability(s2_sh);
    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability, s2_shareability);
    memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    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);

    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 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);
    if HaveCommonNotPrivateTransExt() 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;
    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[];
    tlbcontext.asid   = if TCR_EL1.A1 == '0' then TTBR0_EL1.ASID else TTBR1_EL1.ASID;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() 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 HaveCommonNotPrivateTransExt() 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;
    tlbcontext.asid   = if TCR_EL2.A1 == '0' then TTBR0_EL2.ASID else TTBR1_EL2.ASID;
    tlbcontext.tg     = tg;
    tlbcontext.ia     = va;

    if HaveCommonNotPrivateTransExt() 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 HaveCommonNotPrivateTransExt() then TTBR0_EL3.CnP else '0';

    return tlbcontext;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.AccessUsesEL

```
// AArch64.AccessUsesEL()
// =====
// Returns the Exception Level of the regime that will manage the translation for a given access type.

bits(2) AArch64.AccessUsesEL(AccType acctype)
    if acctype IN {AccType\_UNPRIV, AccType\_UNPRIVSTREAM} then
        return EL0;
    elsif acctype == AccType\_NV2REGISTER then
        return EL2;
    else
        return PSTATE.EL;
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.FaultAllowsSetAccessFlag

```
// AArch64.FaultAllowsSetAccessFlag()
// =====
// Determine whether the access flag could be set by HW given the fault status

boolean AArch64.FaultAllowsSetAccessFlag(FaultRecord fault)
    if fault.statuscode == Fault\_None then
        return TRUE;
    elsif fault.statuscode IN {Fault\_Alignment, Fault\_Permission} then
        return ConstrainUnpredictable(Unpredictable\_AFUPDATE) == Constraint\_TRUE;
    else
        return FALSE;
```

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, AccType acctype,
                                       boolean iswrite, boolean aligned)

    fault = NoFault();
    fault.acctype = acctype;
    fault.write   = iswrite;

    ispriv = PSTATE.EL != EL0 && !(acctype IN {AccType\_UNPRIV, AccType\_UNPRIVSTREAM});
    regime = TranslationRegime(PSTATE.EL, acctype);
    ss = SecurityStateAtEL(PSTATE.EL);

    AddressDescriptor ipa;
    (fault, ipa) = AArch64.S1Translate(fault, regime, ss, va, acctype, aligned, iswrite, ispriv);

    if fault.statuscode != Fault\_None then
        return CreateFaultyAddressDescriptor(va, fault);

    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        s2fslwalk = FALSE;
        AddressDescriptor pa;
        (fault, pa) = AArch64.S2Translate(fault, ipa, slaarch64, ss, s2fslwalk,
                                         acctype, aligned, iswrite, ispriv);

        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(64)) AArch64.MemSwapTableDesc(FaultRecord fault_in, bits(64) prev_desc,
                                                  bits(64) new_desc, bit ee,
                                                  AddressDescriptor descupdateaddress)
    descupdateaccess = CreateAccessDescriptor(AccType_ATOMICRW);
    FaultRecord fault = fault_in;
    boolean iswrite;

    // All observers in the shareability domain observe the
    // following memory read and write accesses atomically.
    (memstatus, mem_desc) = PhysMemRead(descupdateaddress, 8, descupdateaccess);
    if ee == '1' then
        mem_desc = BigEndianReverse(mem_desc);

    if IsFault(memstatus) then
        iswrite = FALSE;
        fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                       8, fault);

        if IsFault(fault.statuscode) then
            fault.acctype = AccType_ATOMICRW;
            return (fault, bits(64) UNKNOWN);

    if mem_desc == prev_desc then
        ordered_new_desc = if ee == '1' then BigEndianReverse(new_desc) else new_desc;
        memstatus = PhysMemWrite(descupdateaddress, 8, descupdateaccess, ordered_new_desc);

        if IsFault(memstatus) then
            iswrite = TRUE;
            fault = HandleExternalTTWAbort(memstatus, iswrite, descupdateaddress, descupdateaccess,
                                           8, fault);

            fault.acctype = memstatus.acctype;
            if IsFault(fault.statuscode) then
                fault.acctype = AccType_ATOMICRW;
                return (fault, bits(64) UNKNOWN);

    // Reflect what is now in memory (in little endian format)
    mem_desc = new_desc;

    return (fault, mem_desc);
```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.S1DisabledOutput

```
// AArch64.S1DisabledOutput()
// =====
// Map the the VA to IPA/PA and assign default memory attributes
(FaultRecord, AddressDescriptor) AArch64.S1DisabledOutput(FaultRecord fault_in, Regime regime,
                                                         SecurityState ss, bits(64) va,
                                                         AccType acctype, boolean aligned)

walkparams = AArch64.GetS1TTWParams(regime, 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<51:0>;
case ss of
    when SS_Secure    oa.paspace = PAS_Secure;
    when SS_NonSecure oa.paspace = PAS_NonSecure;

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;
    memattrs.tagged = walkparams.dct == '1';
    memattrs.xs = '0';
elseif acctype == AccType_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.tagged = FALSE;
    memattrs.xs = '1';
else
    memattrs.memtype = MemType_Device;
    memattrs.device = DeviceType_nGnRnE;
    memattrs.shareability = Shareability_OSH;
    memattrs.xs = '1';

fault.level = 0;
addrtop = AArch64.AddrTop(walkparams.tbid, acctype, walkparams.tbi);
if !IsZero(va<addrtop:AArch64.PAMax()>) then
    fault.statuscode = Fault_AddressSize;
elseif AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsmid, memattrs) then
    fault.statuscode = Fault_Alignment;

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);
else
    ipa = CreateAddressDescriptor(va, oa, memattrs);
    return (fault, ipa);
```



```

// AArch64.S1Translate()
// =====
// Translate VA to IPA/PA depending on the regime

(FaultRecord, AddressDescriptor) AArch64.S1Translate(FaultRecord fault_in, Regime regime,
                                                    SecurityState ss, bits(64) va,
                                                    AccType acctype, boolean aligned_in,
                                                    boolean iswrite_in, boolean ispriv)

    FaultRecord fault = fault_in;
    boolean aligned = aligned_in;
    boolean iswrite = iswrite_in;
    // Prepare fault fields in case a fault is detected
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    if !AArch64.S1Enabled(regime) then
        return AArch64.S1DisabledOutput(fault, regime, ss, va, acctype, aligned);

    walkparams = AArch64.GetS1TTWParams(regime, va);

    if (AArch64.S1InvalidTxSZ(walkparams) ||
        (!ispriv && walkparams.e0pd == '1') ||
        AArch64.VAIsOutOfRange(va, acctype, regime, walkparams)) then
        fault.statuscode = Fault_Translation;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN);

    AddressDescriptor descaddress;
    TTWState walkstate;
    bits(64) descriptor;
    bits(64) new_desc;
    bits(64) mem_desc;
    repeat
        (fault, descaddress, walkstate, descriptor) = AArch64.S1Walk(fault, walkparams, va, regime,
                                                                    ss, acctype, iswrite, ispriv);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN);

    if acctype == AccType_IFETCH then
        // Flag the fetched instruction is from a guarded page
        SetInGuardedPage(walkstate.guardedpage == '1');

    if AArch64.S1HasAlignmentFault(acctype, aligned, walkparams.ntlsm,
                                    walkstate.memattrs) then
        fault.statuscode = Fault_Alignment;
    elseif IsAtomicRW(acctype) then
        if AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                         ispriv, acctype, FALSE) then
            // The Permission fault was not caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = FALSE;
        elseif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                             ispriv, acctype, TRUE) then
            // The Permission fault was caused by lack of write permissions
            fault.statuscode = Fault_Permission;
            fault.write = TRUE;
    elseif AArch64.S1HasPermissionsFault(regime, ss, walkstate, walkparams,
                                         ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

    new_desc = descriptor;
    if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(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 (fault.statuscode == Fault\_None &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    descriptor<51> == '1' && // Descriptor DBM bit
    (IsAtomicRW(acctype) || iswrite) &&
    !(acctype IN {AccType\_AT, AccType\_ATPAN, AccType\_IC, AccType\_DC})) then
// Clear descriptor AP[2] bit permitting stage 1 writes
new_desc<7> = '0';

AddressDescriptor descupdateaddress;
FaultRecord s2fault;
// Either the access flag was clear or AP<2> is set
if new_desc != descriptor then
    if regime == Regime\_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        s2fslwalk = TRUE;
        aligned = TRUE;
        iswrite = TRUE;
        (s2fault, descupdateaddress) = AArch64.S2Translate(fault, descaddress, slaarch64,
                                                    ss, s2fslwalk, AccType\_ATOMICRW,
                                                    aligned, iswrite, ispriv);

        if s2fault.statuscode != Fault\_None then
            return (s2fault, AddressDescriptor UNKNOWN);
    else
        descupdateaddress = descaddress;

        (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                    walkparams.ee, descupdateaddress);

until new_desc == descriptor || mem_desc == new_desc;

if fault.statuscode != Fault\_None then
    return (fault, AddressDescriptor UNKNOWN);

// Output Address
oa = Stage0A(va, walkparams.tgx, walkstate);
MemoryAttributes memattrs;
if (acctype == AccType\_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 (acctype != AccType\_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
if HaveMTE2Ext() && walkstate.memattrs.tagged then
    memattrs.tagged = ConstrainUnpredictableBool(Unpredictable\_S1CTAGGED);
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);

if acctype == AccType\_ATOMICLS64 && memattrs.memtype == MemType\_Normal then
    if memattrs.inner.attrs != MemAttr\_NC || memattrs.outer.attrs != MemAttr\_NC then
        fault.statuscode = Fault\_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

```

```
ipa = CreateAddressDescriptor(va, oa, memattrs);  
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, SecurityState ss,
                                                    boolean s2fslwalk, AccType acctype,
                                                    boolean aligned, boolean iswrite,
                                                    boolean ispriv)
walkparams = AArch64.GetS2TTWParams(ss, ipa.paddress.paspace, slaarch64);
FaultRecord fault = fault_in;

// Prepare fault fields in case a fault is detected
fault.statuscode = Fault_None; // Ignore any faults from stage 1
fault.secondstage = TRUE;
fault.s2fslwalk = s2fslwalk;
fault.ipaddress = ipa.paddress;

if walkparams.vm != '1' then
    // Stage 2 translation is disabled
    return (fault, ipa);

if (AArch64.S2InvalidTxSZ(walkparams, slaarch64) ||
    AArch64.S2InvalidSL(walkparams) ||
    AArch64.S2InconsistentSL(walkparams) ||
    AArch64.IPAIsOutOfRange(ipa.paddress.address, walkparams)) then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN);

AddressDescriptor descaddress;
TTWState walkstate;
bits(64) descriptor;
bits(64) new_desc;
bits(64) mem_desc;
repeat
    (fault, descaddress, walkstate, descriptor) = AArch64.S2Walk(fault, ipa, walkparams, ss,
                                                                acctype, iswrite, slaarch64);

if fault.statuscode != Fault_None then
    return (fault, AddressDescriptor UNKNOWN);

if AArch64.S2HasAlignmentFault(acctype, aligned, walkstate.memattrs) then
    fault.statuscode = Fault_Alignment;
elseif IsAtomicRW(acctype) then
    if AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
                                     ispriv, acctype, FALSE) then
        // The Permission fault was not caused by lack of write permissions
        fault.statuscode = Fault_Permission;
        fault.write = FALSE;
    elseif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
                                         ispriv, acctype, TRUE) then
        // The Permission fault was caused by lack of write permissions.
        // However, HW updates, which are atomic writes for stage 1
        // descriptors, permissions fault reflect the original access.
        fault.statuscode = Fault_Permission;
        if !fault.s2fslwalk then
            fault.write = TRUE;
    elseif AArch64.S2HasPermissionsFault(s2fslwalk, walkstate, ss, walkparams,
                                         ispriv, acctype, iswrite) then
        fault.statuscode = Fault_Permission;

new_desc = descriptor;
if walkparams.ha == '1' && AArch64.FaultAllowsSetAccessFlag(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 (fault.statuscode == Fault\_None &&
    walkparams.ha == '1' &&
    walkparams.hd == '1' &&
    descriptor<51> == '1' && // Descriptor DBM bit
    (IsAtomicRW(acctype) || iswrite) &&
    !(acctype IN {AccType\_AT, AccType\_ATPAN, AccType\_IC, AccType\_DC})) then
// Set descriptor S2AP[1] bit permitting stage 2 writes
new_desc<7> = '1';

// Either the access flag was clear or S2AP<1> is clear
if new_desc != descriptor then
    (fault, mem_desc) = AArch64.MemSwapTableDesc(fault, descriptor, new_desc,
                                                walkparams.ee, descaddress);

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 = Stage0A(ipa_64, walkparams.tgx, walkstate);
MemoryAttributes s2_memattrs;
if ((s2fslwalk &&
    walkstate.memattrs.memtype == MemType\_Device && walkparams.ptw == '0') ||
    (acctype == AccType\_IFETCH &&
    (walkstate.memattrs.memtype == MemType\_Device || HCR_EL2.ID == '1')) ||
    (acctype != AccType\_IFETCH &&
    walkstate.memattrs.memtype == MemType\_Normal && HCR_EL2.CD == '1')) then
// Treat memory attributes as Normal Non-Cacheable
s2_memattrs = NormalNCMemAttr();
s2_memattrs.xs = walkstate.memattrs.xs;
else
    s2_memattrs = walkstate.memattrs;

if !s2fslwalk && acctype == AccType\_ATOMICLS64 && s2_memattrs.memtype == MemType\_Normal then
    if s2_memattrs.inner.attrs != MemAttr\_NC || s2_memattrs.outer.attrs != MemAttr\_NC then
        fault.statuscode = Fault\_Exclusive;
        return (fault, AddressDescriptor UNKNOWN);

MemoryAttributes memattrs;
if walkparams.fwb == '0' then
    memattrs = S2CombineS1MemAttrs(ipa.memattrs, s2_memattrs);
else
    memattrs = s2_memattrs;

pa = CreateAddressDescriptor(ipa.vaddress, oa, memattrs);
return (fault, pa);

```

Library pseudocode for aarch64/translation/vmsa_translation/AArch64.TranslateAddress

```

// AArch64.TranslateAddress()
// =====
// Main entry point for translating an address

AddressDescriptor AArch64.TranslateAddress(bits(64) va, AccType acctype, boolean iswrite,
                                          boolean aligned, integer size)
    result = AArch64.FullTranslate(va, acctype, iswrite, aligned);

if !IsFault(result) then
    result.fault = AArch64.CheckDebug(va, acctype, iswrite, size);

// Update virtual address for abort functions
result.vaddress = ZeroExtend(va);

return result;

```

Library pseudocode for aarch64/translation/vmsa_tentry/AArch64.BlockDescSupported

```
// AArch64.BlockDescSupported()
// =====
// Determine whether a block descriptor is valid for the given granule size
// and level

boolean AArch64.BlockDescSupported(bit ds, TGx tgx, integer level)
  case tgx of
    when TGx\_4KB return level == 2 || level == 1 || (level == 0 && ds == '1');
    when TGx\_16KB return level == 2 || (level == 1 && ds == '1');
    when TGx\_64KB return level == 2 || (level == 1 && AArch64.PAMax() == 52);

  return FALSE;
```

Library pseudocode for aarch64/translation/vmsa_tentry/AArch64.BlocknTFaults

```
// AArch64.BlocknTFaults()
// =====
// Identify whether the nT bit in a block descriptor is effectively set
// causing a translation fault

boolean AArch64.BlocknTFaults(bits(64) descriptor)
  if !HaveBlockBBM() then
    return FALSE;

  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} && descriptor<16> == '1' && nT_faults;
```

Library pseudocode for aarch64/translation/vmsa_tentry/AArch64.ContiguousBit

```
// AArch64.ContiguousBit()
// =====
// Get the value of the contiguous bit

bit AArch64.ContiguousBit(TGx tgx, integer level, bits(64) descriptor)
  if tgx == TGx\_64KB && level == 1 && !Have52BitVAExt() then
    return '0'; // RES0
  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_tentry/AArch64.DecodeDescriptorType

```
// AArch64.DecodeDescriptorType()
// =====
// Determine whether the descriptor is a page, block or table

DescriptorType AArch64.DecodeDescriptorType(bits(64) descriptor, bit ds,
TGx tgx, integer level)
  if descriptor<1:0> == '11' && level == FINAL\_LEVEL then
    return DescriptorType\_Page;
  elsif descriptor<1:0> == '11' then
    return DescriptorType\_Table;
  elsif descriptor<1:0> == '01' then
    if AArch64.BlockDescSupported(ds, tgx, level) then
      return DescriptorType\_Block;
    else
      return DescriptorType\_Invalid;
  else
    return DescriptorType\_Invalid;
```

Library pseudocode for aarch64/translation/vmsa_tentry/AArch64.S1ApplyOutputPerms

```
// AArch64.S1ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 1 page/block descriptors

Permissions AArch64.S1ApplyOutputPerms(Permissions permissions_in, bits(64) descriptor,
                                         Regime regime, S1TTWParams walkparams)
    Permissions permissions = permissions_in;
    if regime == Regime_EL10 && EL2Enabled() && walkparams.nv1 == '1' then
        permissions.ap<2:1> = descriptor<7>:'0';
        permissions.pxn      = descriptor<54>;
    elsif HasUnprivileged(regime) then
        permissions.ap<2:1> = descriptor<7:6>;
        permissions.uxn      = descriptor<54>;
        permissions.pxn      = descriptor<53>;
    else
        permissions.ap<2:1> = descriptor<7>:'1';
        permissions.xn      = descriptor<54>;

    // 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 walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.ap<2> = '0';

    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 uxnx_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>;
        uxnx_table = descriptor<60>;
        pxn_table = descriptor<59>;
        permissions.ap_table = permissions.ap_table OR ap_table;
        permissions.uxnx_table = permissions.uxnx_table OR uxnx_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_tentry/AArch64.S2ApplyOutputPerms

```
// AArch64.S2ApplyOutputPerms()
// =====
// Apply output permissions encoded in stage 2 page/block descriptors

Permissions AArch64.S2ApplyOutputPerms(bits(64) descriptor, S2TTWParams walkparams)
    Permissions permissions;

    permissions.s2ap = descriptor<7:6>;
    permissions.s2xn = descriptor<54>;

    if HaveExtendedExecuteNeverExt() then
        permissions.s2xnx = descriptor<53>;
    else
        permissions.s2xnx = '0';

    // 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.
    if walkparams.ha == '1' && walkparams.hd == '1' && descriptor<51> == '1' then
        permissions.s2ap<1> = '1';

    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;

    startlevel = AArch64.S1StartLevel(walkparams);
    ttbr = AArch64.S1TTBR(regime, va);
    case ss of
        when SS\_Secure tablebase.paspace = PAS\_Secure;
        when SS\_NonSecure tablebase.paspace = PAS\_NonSecure;

    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
        walkparams.tgx, startlevel);

    permissions.ap_table = Zeros();
    if HasUnprivileged(regime) then
        permissions.uxn_table = Zeros();
        permissions.pxn_table = Zeros();
    else
        permissions.xn_table = Zeros();

    walkstate.baseaddress = tablebase;
    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 = permissions;

    return walkstate;
```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S1NextWalkStateLast

```
// AArch64.S1NextWalkStateLast()
// =====
// Decode stage 1 page or block descriptor as output to this stage of translation

TTWState AArch64.S1NextWalkStateLast(TTWState currentstate, Regime regime, SecurityState ss,
                                     S1TTWParams walkparams, bits(64) descriptor)
    TTWState    nextstate;
    FullAddress baseaddress;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, walkparams.ds, walkparams.tgx,
                                                currentstate.level);

    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the block from NS bit
        baseaddress.paspace = if descriptor<5> == '0' then PAS_Secure else PAS_NonSecure;
    else
        baseaddress.paspace = PAS_NonSecure;

    nextstate.istable    = FALSE;
    nextstate.level     = currentstate.level;
    nextstate.baseaddress = baseaddress;

    attrindx = descriptor<4:2>;
    sh = if walkparams.ds == '1' then walkparams.sh else descriptor<9:8>;
    attr = MAIRAttr(UInt(attrindx), walkparams.mair);
    slaarch64 = TRUE;

    nextstate.memattrs    = S1DecodeMemAttrs(attr, sh, slaarch64);
    nextstate.permissions = AArch64.S1ApplyOutputPerms(currentstate.permissions, descriptor,
                                                       regime, walkparams);
    nextstate.contiguous  = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    if !HasUnprivileged(regime) then
        nextstate.nG = '0';
    elsif 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';
    else
        nextstate.nG = descriptor<11>;

    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, Regime regime, S1TTWParams walkparams,
                                       bits(64) descriptor)
    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    if currentstate.baseaddress.paspace == PAS_Secure then
        // Determine PA space of the next table from NSTable bit
        tablebase.paspace = if descriptor<63> == '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;
    nextstate.level     = currentstate.level + 1;
    nextstate.baseaddress = tablebase;
    nextstate.memattrs  = currentstate.memattrs;

    if walkparams.hpd == '0' then
        nextstate.permissions = AArch64.S1ApplyTablePerms(currentstate.permissions, descriptor,
                                                         regime, walkparams);
    else
        nextstate.permissions = currentstate.permissions;

    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(64)) AArch64.S1Walk(FaultRecord fault_in,
    S1TTWParams walkparams, bits(64) va, Regime regime, SecurityState ss,
    AccType acctype, boolean iswrite_in, boolean ispriv)
FaultRecord fault = fault_in;
boolean iswrite = iswrite_in;
if HasUnprivileged(regime) && AArch64.S1EPD(regime, va) == '1' then
    fault.statuscode = Fault_Translation;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

walkstate = AArch64.S1InitialTTWState(walkparams, va, regime, ss);

// Detect Address Size Fault by TTB
if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
    fault.statuscode = Fault_AddressSize;
    fault.level = 0;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

bits(64) descriptor;
AddressDescriptor walkaddress;

walkaddress.vaddress = va;
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);

DescriptorType descctype;
repeat
    fault.level = walkstate.level;
    FullAddress descaddress = AArch64.TTEntryAddress(walkstate.level, walkparams.tgx,
        walkparams.txs, va, walkstate.baseaddress);

    walkaddress.paddress = descaddress;

    if regime == Regime_EL10 && EL2Enabled() then
        slaarch64 = TRUE;
        s2fslwalk = TRUE;
        aligned = TRUE;
        iswrite = FALSE;
        (s2fault, s2walkaddress) = AArch64.S2Translate(fault, walkaddress, slaarch64, ss,
            s2fslwalk, AccType_TTW, aligned, iswrite, ispriv);

        if s2fault.statuscode != Fault_None then
            return (s2fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        (fault, descriptor) = FetchDescriptor(walkparams.ee, s2walkaddress, fault);
    else
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

    if fault.statuscode != Fault_None then
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    descctype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,

```

```

                                walkstate.level);

case desctype of
  when DescriptorType\_Table
    walkstate = AArch64.S1NextWalkStateTable(walkstate, regime, walkparams,
                                              descriptor);

    // Detect Address Size Fault by table descriptor
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
      fault.statuscode = Fault\_AddressSize;
      return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

  when DescriptorType\_Page, DescriptorType\_Block
    walkstate = AArch64.S1NextWalkStateLast(walkstate, regime, ss,
                                             walkparams, descriptor);

  when DescriptorType\_Invalid
    fault.statuscode = Fault\_Translation;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

  otherwise
    Unreachable();

until desctype IN {DescriptorType\_Page, DescriptorType\_Block};

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
  fault.statuscode = Fault\_Translation;
elseif desctype == DescriptorType\_Block && AArch64.BlocknTFaults(descriptor) then
  fault.statuscode = Fault\_Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, va) then
  fault.statuscode = Fault\_AddressSize;
// Check descriptor AF bit
elseif (descriptor<10> == '0' && walkparams.ha == '0' &&
        !(acctype IN {AccType\_DC, AccType\_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.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;

  ttbr = VTTBR_EL2;
  startlevel = AArch64.S2StartLevel(walkparams);
  tablebase.paspace = PAS\_NonSecure;
  tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
                                           walkparams.tgx, startlevel);

  walkstate.baseaddress = tablebase;
  walkstate.level = startlevel;
  walkstate.istable = TRUE;
  walkstate.memattrs = WalkMemAttrs(walkparams.sh, walkparams.irgn, walkparams.orgn);

return walkstate;

```

Library pseudocode for aarch64/translation/vmsa_walk/AArch64.S2NextWalkStateLast

```
// AArch64.S2NextWalkStateLast()
// =====
// Decode stage 2 page or block descriptor as output to this stage of translation

TTWState AArch64.S2NextWalkStateLast(TTWState currentstate, SecurityState ss,
                                     S2TTWParams walkparams, AddressDescriptor ipa,
                                     bits(64) descriptor)

    TTWState    nextstate;
    FullAddress baseaddress;

    if ss == SS_Secure then
        baseaddress.paspace = AArch64.SS20outputPASpace(walkparams, ipa.paddress.paspace);
    else
        baseaddress.paspace = PAS_NonSecure;

    if currentstate.level == FINAL_LEVEL then
        baseaddress.address = AArch64.PageBase(descriptor, walkparams.ds, walkparams.tgx);
    else
        baseaddress.address = AArch64.BlockBase(descriptor, 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, s2_attr, s2_sh);
        if s2_attr<1:0> == '10' then // Force writeback
            nextstate.memattrs.xs = '0';
        else
            nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    else
        nextstate.memattrs = S2DecodeMemAttrs(s2_attr, s2_sh);
        nextstate.memattrs.xs = if s2_fnxs == '1' then '0' else ipa.memattrs.xs;
    nextstate.contiguous = AArch64.ContiguousBit(walkparams.tgx, currentstate.level, descriptor);

    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(64) descriptor)

    TTWState    nextstate;
    FullAddress tablebase;

    tablebase.address = AArch64.NextTableBase(descriptor, walkparams.ds, walkparams.tgx);
    tablebase.paspace = currentstate.baseaddress.paspace;

    nextstate.istable    = TRUE;
    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(64)) AArch64.S2Walk(
    FaultRecord fault_in, AddressDescriptor ipa, S2TTWParams walkparams, SecurityState ss,
    AccType acctype, boolean iswrite, boolean slaarch64)

    FaultRecord fault = fault_in;
    ipa_64 = ZeroExtend(ipa.paddress.address, 64);

    TTWState walkstate;
    if ss == SS_Secure then
        walkstate = AArch64.SS2InitialTTWState(walkparams, ipa.paddress.paspace);
    else
        walkstate = AArch64.S2InitialTTWState(ss, walkparams);

    // Detect Address Size Fault by TTB
    if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
        fault.statuscode = Fault_AddressSize;
        fault.level = 0;
        return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

    bits(64) descriptor;
    AddressDescriptor walkaddress;

    walkaddress.vaddress = ipa.vaddress;
    if HCR_EL2.CD == '1' then
        walkaddress.memattrs = NormalNCMemAttr();
        walkaddress.memattrs.xs = walkstate.memattrs.xs;
    else
        walkaddress.memattrs = walkstate.memattrs;

    walkaddress.memattrs.shareability = EffectiveShareability(walkaddress.memattrs);

    DescriptorType descstype;
    repeat
        fault.level = walkstate.level;

        FullAddress descaddress;
        if walkstate.level == AArch64.S2StartLevel(walkparams) then
            // Initial lookup might index into concatenated tables
            descaddress = AArch64.S2SLTTEnterAddress(walkparams, ipa.paddress.address,
                walkstate.baseaddress);
        else
            ipa_64 = ZeroExtend(ipa.paddress.address, 64);
            descaddress = AArch64.TTEnterAddress(walkstate.level, walkparams.tgx, walkparams.txsz,
                ipa_64, walkstate.baseaddress);

        walkaddress.paddress = descaddress;
        (fault, descriptor) = FetchDescriptor(walkparams.ee, walkaddress, fault);

        if fault.statuscode != Fault_None then
            return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

        descstype = AArch64.DecodeDescriptorType(descriptor, walkparams.ds, walkparams.tgx,
            walkstate.level);

        case descstype of
            when DescriptorType_Table
                walkstate = AArch64.S2NextWalkStateTable(walkstate, walkparams, descriptor);

                // Detect Address Size Fault by table descriptor
                if AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
                    fault.statuscode = Fault_AddressSize;
                    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

            when DescriptorType_Page, DescriptorType_Block
                walkstate = AArch64.S2NextWalkStateLast(walkstate, ss, walkparams, ipa,

```

```

descriptor);

when DescriptorType_Invalid
    fault.statuscode = Fault_Translation;
    return (fault, AddressDescriptor UNKNOWN, TTWState UNKNOWN, bits(64) UNKNOWN);

otherwise
    Unreachable();

until descctype IN {DescriptorType_Page, DescriptorType_Block};

if (walkstate.contiguous == '1' &&
    AArch64.ContiguousBitFaults(walkparams.txsz, walkparams.tgx, walkstate.level)) then
    fault.statuscode = Fault_Translation;
elseif descctype == DescriptorType_Block && AArch64.BlocknTFaults(descriptor) then
    fault.statuscode = Fault_Translation;
// Detect Address Size Fault by final output
elseif AArch64.OAOutOfRange(walkstate, walkparams.ps, walkparams.tgx, ipa_64) then
    fault.statuscode = Fault_AddressSize;
// Check descriptor AF bit
elseif (descriptor<10> == '0' && walkparams.ha == '0' &&
        !(acctype IN {AccType_DC, AccType_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(64) ttbr;
    if ipaspace == PAS_Secure then
        ttbr = VSTTBR_EL2;
    else
        ttbr = VTTBR_EL2;

    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;

    startlevel = AArch64.S2StartLevel(walkparams);
    tablebase.address = AArch64.TTBaseAddress(ttbr, walkparams.txsz, walkparams.ps, walkparams.ds,
        walkparams.tgx, startlevel);

    walkstate.baseaddress = tablebase;
    walkstate.level = startlevel;
    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 !HaveBlockBBM() then
        return integer UNKNOWN;
    else
        return integer IMPLEMENTATION_DEFINED "Block BBM support level";
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.DecodeTG0

```
// AArch64.DecodeTG0()
// =====
// Decode granule size configuration bits TG0

TGx AArch64.DecodeTG0(bits(2) tg0_in)
    bits(2) tg0 = tg0_in;
    if tg0 == '11' then
        tg0 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG0 encoding granule size";

    case tg0 of
        when '00' return TGx_4KB;
        when '01' return TGx_64KB;
        when '10' return TGx_16KB;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.DecodeTG1

```
// AArch64.DecodeTG1()
// =====
// Decode granule size configuration bits TG1

TGx AArch64.DecodeTG1(bits(2) tg1_in)
    bits(2) tg1 = tg1_in;
    if tg1 == '00' then
        tg1 = bits(2) IMPLEMENTATION_DEFINED "Reserved TG1 encoding granule size";

    case tg1 of
        when '10' return TGx_4KB;
        when '11' return TGx_64KB;
        when '01' return TGx_16KB;
```

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, bits(64) va)
    S1TTWParams walkparams;

    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3 walkparams = AArch64.S1TTWParamsEL3();
        when Regime_EL2 walkparams = AArch64.S1TTWParamsEL2();
        when Regime_EL20 walkparams = AArch64.S1TTWParamsEL20(varange);
        when Regime_EL10 walkparams = AArch64.S1TTWParamsEL10(varange);

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S1MinTxSZ(walkparams.ds, walkparams.tgx);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elseif !Have52BitVAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    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);
    elseif HaveSecureEL2Ext() && ss == SS_Secure then
        walkparams = AArch64.SS2TTWParams(ipaspace, slaarch64);
    else
        Unreachable();

    maxtxsz = AArch64.MaxTxSZ(walkparams.tgx);
    mintxsz = AArch64.S2MinTxSZ(walkparams.ds, walkparams.tgx, slaarch64);
    if UInt(walkparams.txsz) > maxtxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value above maximum") then
            walkparams.txsz = maxtxsz<5:0>;
    elseif !Have52BitPAExt() && UInt(walkparams.txsz) < mintxsz then
        if !(boolean IMPLEMENTATION_DEFINED "Fault on TxSZ value below minimum") then
            walkparams.txsz = mintxsz<5:0>;

    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.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 HaveSmallTranslationTableExt() && !UsingAArch32() then
        case tgx of
            when TGx_4KB    return 48;
            when TGx_16KB   return 48;
            when TGx_64KB   return 47;
        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.DecodeTG0(VTCR_EL2.TG0);
    walkparams.txsz = VTCR_EL2.T0SZ;
    walkparams.sl0 = VTCR_EL2.SL0;
    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 HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
        walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
    else
        walkparams.sl2 = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '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

integer AArch64.PAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Physical Address Size";
```

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 SCTLRL_EL3.C == '1';
        when Regime_EL2 return SCTLRL_EL2.C == '1';
        when Regime_EL20 return SCTLRL_EL2.C == '1';
        when Regime_EL10 return SCTLRL_EL1.C == '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 for the acting translation regime is enabled

boolean AArch64.S1Enabled(Regime regime)
    case regime of
        when Regime_EL3 return SCTLRL_EL3.M == '1';
        when Regime_EL2 return SCTLRL_EL2.M == '1';
        when Regime_EL20 return SCTLRL_EL2.M == '1';
        when Regime_EL10 return (!EL2Enabled() || HCR_EL2.<DC,TGE> == '00') && SCTLRL_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 SCTLRL_EL3.I == '1';
        when Regime_EL2 return SCTLRL_EL2.I == '1';
        when Regime_EL20 return SCTLRL_EL2.I == '1';
        when Regime_EL10 return SCTLRL_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(bit ds, TGx tgx)
    if (Have52BitVAExt() && tgx == TGx_64KB) || ds == '1' then
        return 12;

    return 16;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTBR

```
// AArch64.S1TTBR()
// =====
// Identify stage 1 table base register for the acting translation regime

bits(64) AArch64.S1TTBR(Regime regime, bits(64) va)
    varange = AArch64.GetVARange(va);

    case regime of
        when Regime_EL3 return TTBR0_EL3;
        when Regime_EL2 return TTBR0_EL2;
        when Regime_EL20 return if varange == VARange_LOWER then TTBR0_EL2 else TTBR1_EL2;
        when Regime_EL10 return if varange == VARange_LOWER then TTBR0_EL1 else TTBR1_EL1;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL10

```
// 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 varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(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.tbid = if HavePACEExt() then TCR_EL1.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD0 else '0';
        walkparams.hpdpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL1.TG1);
        walkparams.txsz = TCR_EL1.T1SZ;
        walkparams.irgn = TCR_EL1.IRGN1;
        walkparams.orgn = TCR_EL1.ORGNO;
        walkparams.sh = TCR_EL1.SH1;
        walkparams.tbi = TCR_EL1.TBI1;

        walkparams.tbid = if HavePACEExt() then TCR_EL1.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL1.E0PD1 else '0';
        walkparams.hpdpd = if AArch64.HaveHPDEExt() then TCR_EL1.HPD1 else '0';

    walkparams.mair = MAIR_EL1;
    walkparams.wxn = SCTL_EL1.WXN;
    walkparams.ps = TCR_EL1.IPS;
    walkparams.ee = SCTL_EL1.EE;
    walkparams.sif = SCR_EL3.SIF;

    if EL2Enabled() then
        walkparams.dc = HCR_EL2.DC;
        walkparams.dct = if HaveMTE2Ext() then HCR_EL2.DCT else '0';

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsm = SCTL_EL1.nTlsm;
    else
        walkparams.ntlsm = '1';

    if EL2Enabled() then
        if HCR_EL2.<NV,NV1> == '01' then
            case ConstrainUnpredictable(Unpredictable_NVNV1) of
                when Constraint_NVNV1_00 walkparams.nv1 = '0';
                when Constraint_NVNV1_01 walkparams.nv1 = '1';
                when Constraint_NVNV1_11 walkparams.nv1 = '1';
            else
                walkparams.nv1 = HCR_EL2.NV1;
        else
            walkparams.nv1 = '0';

    walkparams.epan = if HavePAN3Ext() then SCTL_EL1.EPAN else '0';
    walkparams.cmow = if HaveFeatCMOW() then SCTL_EL1.CMOW else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL1.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL1.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAndPASpaceExt() then
        walkparams.ds = TCR_EL1.DS;
    else
        walkparams.ds = '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()
    S1TTWParams walkparams;

    walkparams.tgx = AArch64.DecodeTG0(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.wxn = SCTLR_EL2.WXN;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt() then TCR_EL2.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL2.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.S1TTWParamsEL20

```
// AArch64.S1TTWParamsEL20()
// =====
// Gather stage 1 translation table walk parameters for EL2&0 regime

S1TTWParams AArch64.S1TTWParamsEL20(VARange varange)
    S1TTWParams walkparams;

    if varange == VARange_LOWER then
        walkparams.tgx = AArch64.DecodeTG0(TCR_EL2.TG0);
        walkparams.txsz = TCR_EL2.T0SZ;
        walkparams.irgn = TCR_EL2.IRGN0;
        walkparams.orgn = TCR_EL2.ORGNO;
        walkparams.sh = TCR_EL2.SH0;
        walkparams.tbi = TCR_EL2.TBI0;

        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID0 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD0 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD0 else '0';
    else
        walkparams.tgx = AArch64.DecodeTG1(TCR_EL2.TG1);
        walkparams.txsz = TCR_EL2.T1SZ;
        walkparams.irgn = TCR_EL2.IRGN1;
        walkparams.orgn = TCR_EL2.ORGNI;
        walkparams.sh = TCR_EL2.SH1;
        walkparams.tbi = TCR_EL2.TBI1;

        walkparams.tbid = if HavePACExt() then TCR_EL2.TBID1 else '0';
        walkparams.e0pd = if HaveE0PDEExt() then TCR_EL2.E0PD1 else '0';
        walkparams.hpd = if AArch64.HaveHPDEExt() then TCR_EL2.HPD1 else '0';

    walkparams.mair = MAIR_EL2;
    walkparams.wxn = SCTLR_EL2.WXN;
    walkparams.ps = TCR_EL2.IPS;
    walkparams.ee = SCTLR_EL2.EE;
    walkparams.sif = SCR_EL3.SIF;

    if HaveTrapLoadStoreMultipleDeviceExt() then
        walkparams.ntlsmid = SCTLR_EL2.nTlSMID;
    else
        walkparams.ntlsmid = '1';

    walkparams.epan = if HavePAN3Ext() then SCTLR_EL2.EPAN else '0';
    walkparams.cmov = if HaveFeatCMOW() then SCTLR_EL2.CMOW else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAndPAspaceExt() then
        walkparams.ds = TCR_EL2.DS;
    else
        walkparams.ds = '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.DecodeTG0(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.wxn = SCTLRL_EL3.WXN;
    walkparams.ee = SCTLRL_EL3.EE;
    walkparams.sif = SCR_EL3.SIF;

    walkparams.tbid = if HavePACExt() then TCR_EL3.TBID else '0';
    walkparams.hpd = if AArch64.HaveHPDExt() then TCR_EL3.HPD else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then TCR_EL3.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then TCR_EL3.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAndPASpaceExt() then
        walkparams.ds = TCR_EL3.DS;
    else
        walkparams.ds = '0';

    return walkparams;
```

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 ds, TGx tgx, boolean slaarch64)
    ips = AArch64.PAMax();

    if Have52BitPAExt() && tgx != TGx_64KB && ds == '0' then
        ips = Min(48, AArch64.PAMax());

    min_txsz = 64 - ips;
    if !slaarch64 then
        // EL1 is AArch32
        min_txsz = Min(min_txsz, 24);

    return min_txsz;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.SS2TTWParams

```
// AArch64.SS2TTWParams()
// =====
// Gather walk parameters specific for secure stage 2 translation
S2TTWParams AArch64.SS2TTWParams(PASpace ipaspace, boolean slaarch64)
    S2TTWParams walkparams;

    if ipaspace == PAS_Secure then
        walkparams.tgx = AArch64.DecodeTG0(VSTCR_EL2.TG0);
        walkparams.txsz = VSTCR_EL2.T0SZ;
        walkparams.sl0 = VSTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VSTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    elseif ipaspace == PAS_NonSecure then
        walkparams.tgx = AArch64.DecodeTG0(VTCR_EL2.TG0);
        walkparams.txsz = VTCR_EL2.T0SZ;
        walkparams.sl0 = VTCR_EL2.SL0;
        if walkparams.tgx == TGx_4KB && Have52BitIPAAAndPASpaceExt() then
            walkparams.sl2 = VTCR_EL2.SL2 AND VTCR_EL2.DS;
        else
            walkparams.sl2 = '0';
    else
        Unreachable();

    walkparams.sw = VSTCR_EL2.SW;
    walkparams.nsw = VTCR_EL2.NSW;
    walkparams.sa = VSTCR_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 HaveStage2MemAttrControl() then HCR_EL2.FWB else '0';
    walkparams.ha = if HaveAccessFlagUpdateExt() then VTCR_EL2.HA else '0';
    walkparams.hd = if HaveDirtyBitModifierExt() then VTCR_EL2.HD else '0';
    if walkparams.tgx IN {TGx_4KB, TGx_16KB} && Have52BitIPAAAndPASpaceExt() then
        walkparams.ds = VTCR_EL2.DS;
    else
        walkparams.ds = '0';
    walkparams.cmow = if HaveFeatCMOW() && IsHCRXEL2Enabled() then HCRX_EL2.CMOW else '0';

    return walkparams;
```

Library pseudocode for aarch64/translation/vmsa_walkparams/AArch64.VAMax

```
// AArch64.VAMax()
// =====
// Returns the IMPLEMENTATION_DEFINED maximum number of bits capable of representing
// the virtual address for this processor

integer AArch64.VAMax()
    return integer IMPLEMENTATION_DEFINED "Maximum Virtual Address Size";
```

Library pseudocode for shared/debug/ClearStickyErrors/ClearStickyErrors

```
// ClearStickyErrors()
// =====

ClearStickyErrors()
    EDSCR.TXU = '0';           // Clear TX underrun flag
    EDSCR.RX0 = '0';           // Clear RX overrun flag

    if Halted() then           // in Debug state
        EDSCR.IT0 = '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 ||
        (HaveSecureEL2Ext() && (!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 !HaveDoubleLock() 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

```
enumeration Component {
    Component_PMU,
    Component_Debug,
    Component_CTI
};
```

Library pseudocode for shared/debug/SoftwareLockStatus/GetAccessComponent

```
// Returns the accessed component.
Component GetAccessComponent();
```

Library pseudocode for shared/debug/SoftwareLockStatus/SoftwareLockStatus

```
// SoftwareLockStatus()
// =====
// Returns the state of the Software Lock.

boolean SoftwareLockStatus()
    Component component = GetAccessComponent();
    if !HaveSoftwareLock(component) then
        return FALSE;
    case component of
        when Component_Debug
            return EDLSR.SLK == '1';
        when Component_PMU
            return PMLSR.SLK == '1';
        when Component_CTI
            return CTILSR.SLK == '1';
    otherwise
        Unreachable();
```

Library pseudocode for shared/debug/authentication/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 HaveSecureExtDebugView() 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/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 HaveSecureExtDebugView() 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/Debug_authentication

```
signal DBGEN;
signal NIDEN;
signal SPIDEN;
signal SPNIDEN;
```

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 == HIGH;
```

Library pseudocode for shared/debug/authentication/ExternalNoninvasiveDebugEnabled

```
// ExternalNoninvasiveDebugEnabled()
// =====
// Returns TRUE if Trace and PC Sample-based Profiling are allowed

boolean ExternalNoninvasiveDebugEnabled()
    if !ExternalInvasiveDebugEnabled() then return FALSE;
    ss = SecurityStateAtEL(PSTATE.EL);
    if (ELUsingAArch32(EL1) && PSTATE.EL == EL0 &&
        ss == SS_Secure && SDER.SUNIDEN == '1') then
        return TRUE;
    case ss of
        when SS_NonSecure return TRUE;
        when SS_Secure    return ExternalSecureNoninvasiveDebugEnabled();
```

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 !HaveNoninvasiveDebugAuth() || ExternalInvasiveDebugEnabled() || NIDEN == HIGH;
```

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 !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    return ExternalInvasiveDebugEnabled() && SPIDEN == 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 !HaveEL(EL3) && !SecureOnlyImplementation() then return FALSE;
    if HaveNoninvasiveDebugAuth() then
        return ExternalNoninvasiveDebugEnabled() && (SPIDEN == HIGH || SPNIDEN == HIGH);
    else
        return ExternalSecureInvasiveDebugEnabled();
```

Library pseudocode for shared/debug/authentication/IsAccessSecure

```
// Returns TRUE when an access is Secure
boolean IsAccessSecure();
```

Library pseudocode for shared/debug/authentication/IsCorePowered

```
// Returns TRUE if the Core power domain is powered on, FALSE otherwise.
boolean IsCorePowered();
```

Library pseudocode for shared/debug/breakpoint/CheckValidStateMatch

```
// CheckValidStateMatch()
// =====
// Checks for an invalid state match that will generate Constrained
// Unpredictable behaviour, otherwise returns Constraint_NONE.
(Constraint, bits(2), bit, bits(2)) CheckValidStateMatch(bits(2) SSC_in, bit HMC_in, bits(2) PxC_in,
boolean isbreakpnt)
    boolean reserved = FALSE;
    bits(2) SSC = SSC_in;
    bit HMC = HMC_in;
    bits(2) PxC = PxC_in;

    // Values that are not allocated in any architecture version
    if (HMC:SSC:PxC) IN {'01110', '100x0', '10110', '11x10'} then
        reserved = TRUE;

    // 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 !HaveSecureEL2Ext() && (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,PxC>) = ConstrainUnpredictableBits(Unpredictable_RESBPWCTRL);
        assert c IN {Constraint_DISABLED, Constraint_UNKNOWN};
        if c == Constraint_DISABLED then
            return (c, bits(2) UNKNOWN, bit UNKNOWN, bits(2) UNKNOWN);
        // Otherwise the value returned by ConstrainUnpredictableBits must be a not-reserved value

    return (Constraint_NONE, SSC, HMC, PxC);
```

Library pseudocode for shared/debug/breakpoint/NumBreakpointsImplemented

```
// NumBreakpointsImplemented()
// =====
// Returns the number of breakpoints implemented. This is indicated to software by
// DBGDIDR.BRPs in AArch32 state, and ID_AA64DFR0_EL1.BRPs in AArch64 state.

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. This is indicated to software by
// DBGDIDR.CTX_CMPs in AArch32 state, and ID_AA64DFR0_EL1.CTX_CMPs in AArch64 state.

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. This is indicated to software by
// DBGDIDR.WRPs in AArch32 state, and ID_AA64DFR0_EL1.WRPs in AArch64 state.

integer NumWatchpointsImplemented()
    return integer IMPLEMENTATION_DEFINED "Number of watchpoints";
```

Library pseudocode for shared/debug/cti/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

```
// Signal a discrete event on a Cross Trigger input event trigger.
CTI_SignalEvent(CrossTriggerIn id);
```

Library pseudocode for shared/debug/cti/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()
  comrx = (EDSCR.RXfull == '1');
  commtx = (EDSCR.TXfull == '0');

  // COMMRX and COMMTX support is optional and not recommended for new designs.
  // SetInterruptRequestLevel(InterruptID_COMMRX, if comrx then HIGH else LOW);
  // SetInterruptRequestLevel(InterruptID_COMMTX, if commtx then HIGH else LOW);

  // The value to be driven onto the common COMMIRQ signal.
  boolean commirq;
  if ELUsingAArch32\(EL1\) then
    commirq = ((comrx && DBGDCCINT.RX == '1') ||
              (commtx && DBGDCCINT.TX == '1'));
  else
    commirq = ((comrx && MDCCINT_EL1.RX == '1') ||
              (commtx && MDCCINT_EL1.TX == '1'));
  SetInterruptRequestLevel(InterruptID\_COMMIRQ, if commirq then HIGH else LOW);

  return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRRX_ELO

```
// DBGDTRRX_ELO[] (external write)
// =====
// Called on writes to debug register 0x08C.

DBGDTRRX_ELO[boolean memory_mapped] = bits(32) value

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

if EDSCR.ERR == '1' then return; // Error flag set: ignore write

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: 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 EDITR[] (external write)
    if !UsingAArch32() then
        ExecuteA64(0xD5330501<31:0>); // A64 "MRS X1,DBGDTRRX_ELO"
        ExecuteA64(0xB8004401<31:0>); // A64 "STR W1,[X0],#4"
        X[1] = 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_ELO = bits(64) UNKNOWN;
    else
        // "MRS X1,DBGDTRRX_ELO" calls DBGDTR_EL0[] (read) which clears RXfull.
        assert EDSCR.RXfull == '0';

    EDSCR.ITE = '1'; // See comments in EDITR[] (external write)
    return;

// DBGDTRRX_ELO[] (external read)
// =====

bits(32) DBGDTRRX_ELO[boolean memory_mapped]
return DTRRX;
```

Library pseudocode for shared/debug/dccanditr/DBGDTRTX_EL0

```
// DBGDTRTX_EL0[] (external read)
// =====
// Called on reads of debug register 0x080.

bits(32) DBGDTRTX_EL0[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

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

// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then // Software lock locked: no side-effects
    return value;

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 // See comments in EDITR[] (external write)
    EDSCR.ITE = '0';

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 DBGDTR_EL0[] (write) which sets TXfull.
    assert EDSCR.TXfull == '1';
if !UsingAArch32() then
    X[1] = bits(64) UNKNOWN;
else
    R[1] = bits(32) UNKNOWN;
EDSCR.ITE = '1'; // See comments in EDITR[] (external write)

return value;

// DBGDTRTX_EL0[] (external write)
// =====

DBGDTRTX_EL0[boolean memory_mapped] = bits(32) value
// The Software lock is OPTIONAL.
if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: ignore write
DTRTX = value;
return;
```

Library pseudocode for shared/debug/dccanditr/DBGDTR_EL0

```
// DBGDTR_EL0[] (write)
// =====
// System register writes to DBGDTR_EL0, DBGDTRTX_EL0 (AArch64) and DBGDTRTXint (AArch32)

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;

// DBGDTR_EL0[] (read)
// =====
// System register reads of DBGDTR_EL0, DBGDTRRX_EL0 (AArch64) and DBGDTRRXint (AArch32)

bits(N) DBGDTR_EL0[]
  // 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/DTR

```
bits(32) DTRRX;
bits(32) DTRTX;
```

Library pseudocode for shared/debug/dccanditr/EDITR

```
// EDITR[] (external write)
// =====
// Called on writes to debug register 0x084.

EDITR[boolean memory_mapped] = bits(32) value
  if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return;

  if EDSCR.ERR == '1' then return; // Error flag set: ignore write

  // The Software lock is OPTIONAL.
  if memory_mapped && EDLSR.SLK == '1' then return; // Software lock locked: 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;
```

Library pseudocode for shared/debug/halting/DCPSInstruction

```

// 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 HavePANExt() && SCTL.R.SPAN == '0' then
                PSTATE.PAN = '1';
        when EL2
            AArch32.WriteMode(M32_Hyp);
        when EL3
            AArch32.WriteMode(M32_Monitor);
            if HavePANExt() then
                if !from_secure then
                    PSTATE.PAN = '0';
                elsif SCTL.R.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[] = bits(32) UNKNOWN;
        PSTATE.E = SCTL.R.EE;
        DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;

else // Targeting AArch64
    if UsingAArch32() then
        AArch64.MaybeZeroRegisterUppers();
    PSTATE.nRW = '0'; PSTATE.SP = '1'; PSTATE.EL = handle_el;
    if HavePANExt() && ((handle_el == EL1 && SCTL.R_EL1.SPAN == '0') ||
        (handle_el == EL2 && HCR_EL2.E2H == '1' &&
        HCR_EL2.TGE == '1' && SCTL.R_EL2.SPAN == '0')) then
        PSTATE.PAN = '1';
    ELR[] = bits(64) UNKNOWN; SPSR[] = bits(64) UNKNOWN; ESR[] = bits(64) UNKNOWN;
    DLR_EL0 = bits(64) UNKNOWN; DSPSR_EL0 = bits(64) UNKNOWN;
    if HaveUAOExt() then PSTATE.UAO = '0';

```

```

    if HaveMTEExt() then PSTATE.TCO = '1';

    UpdateEDSCRFields(); // Update EDSCR PE state flags
    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored 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()

    SynchronizeContext();

    sync_errors = HaveIESB() && SCTLR[].IESB == '1';
    if HaveDoubleFaultExt() && !UsingAArch32() then
        sync_errors = sync_errors || (SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' && PSTATE.EL == EL3);
    // SCTLR[].IESB might be ignored in Debug state.
    if !ConstrainUnpredictableBool(Unpredictable_IESBinDebug) then
        sync_errors = FALSE;
    if sync_errors then
        SynchronizeErrors();

    DebugRestorePSR();

    return;

```

Library pseudocode for shared/debug/halting/DebugHalt

```

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
    bits(32) spsr = SPSR[];
    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
    bits(64) spsr = SPSR[];
    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();
```

Library pseudocode for shared/debug/halting/ExecuteA64

```
// Execute an A64 instruction in Debug state.
ExecuteA64(bits(32) instr);
```

Library pseudocode for shared/debug/halting/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 Havev8p8Debug() then
    EDESR<3:0> = '0000';
  else
    EDESR<2:0> = '000';

  bits(64) new_pc;
  bits(64) spsr;

  if UsingAArch32() then
    new_pc = ZeroExtend(DLR);
    spsr = ZeroExtend(DSPSR);
  else
    new_pc = DLR_EL0;
    spsr = DSPSR_EL0;
  // If this is an illegal return, SetPSTATEFromPSR() will set PSTATE.IL.
  if UsingAArch32() then
    SetPSTATEFromPSR(spsr<31:0>); // Can update privileged bits, even at EL0
  else
    SetPSTATEFromPSR(spsr); // Can update privileged bits, even at EL0

  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 possibly zero the 32 most significant bits of the target PC
    if spsr<4> == '1' && ConstrainUnpredictableBool(Unpredictable_RESTARTZEROUPPERPC) then
      new_pc<63:32> = Zeros();
    // A type of branch that is never predicted
    BranchTo(new_pc, BranchType_DBGEXIT, branch_conditional);

  (EDSCR.STATUS, EDPRSR.SDR) = ('000010', '1'); // Atomically signal restarted
  UpdateEDSCRFields(); // Stop signalling PE state
  DisableITRAndResumeInstructionPrefetch();

return;
```

Library pseudocode for shared/debug/halting/Halt

```
// Halt()
// =====

Halt(bits(6) reason)
    boolean is_async = FALSE;
    Halt(reason, is_async);

// Halt()
// =====

Halt(bits(6) reason, boolean is_async)

    CTI_SignalEvent(CrossTriggerIn_CrossHalt); // Trigger other cores to halt

    bits(64) preferred_restart_address = ThisInstrAddr();
    bits(32) spsr_32;
    bits(64) spsr_64;
    if UsingAArch32() then
        spsr_32 = GetPSRFromPSTATE(DebugState);
    else
        spsr_64 = GetPSRFromPSTATE(DebugState);

    if (HaveBTIExt() && !is_async && !(reason IN {DebugHalt_Step_Normal, DebugHalt_Step_Exclusive,
        DebugHalt_Step_NoSyndrome, DebugHalt_Breakpoint, DebugHalt_HaltInstruction}) &&
        ConstrainUnpredictableBool(Unpredictable_ZEROBTYP)) then
        if UsingAArch32() then
            spsr_32<11:10> = '00';
        else
            spsr_64<11:10> = '00';

    if UsingAArch32() then
        DLR = preferred_restart_address<31:0>;
        DSPSR = spsr_32;
    else
        DLR_EL0 = preferred_restart_address;
        DSPSR_EL0 = spsr_64;

    EDSCR.ITE = '1';
    EDSCR.ITO = '0';
    if CurrentSecurityState() == SS_Secure then
        EDSCR.SDD = '0'; // If entered in Secure state, allow debug
    elsif 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.{UA0,PAN} are observable and not changed on entry into Debug state.
    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;
        PSTATE.TCO = '1';
        PSTATE.BTYPE = '00';
    PSTATE.IL = '0';

    StopInstructionPrefetchAndEnableITR();
    EDSCR.STATUS = reason; // Signal entered Debug state
    UpdateEDSCRFields(); // Update EDSCR PE state flags.
    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();
```

Library pseudocode for shared/debug/halting/Restarting

```
// Restarting()
// =====

boolean Restarting()
    return EDSCR.STATUS == '000001'; // Restarting
```

Library pseudocode for shared/debug/halting/StopInstructionPrefetchAndEnableITR

```
StopInstructionPrefetchAndEnableITR();
```

Library pseudocode for shared/debug/halting/UpdateEDSCRFields

```
// UpdateEDSCRFields()
// =====
// Update EDSCR PE state fields

UpdateEDSCRFields()

  if !Halted() then
    EDSCR.EL = '00';
    EDSCR.NS = bit UNKNOWN;

    EDSCR.RW = '1111';
  else
    EDSCR.EL = PSTATE.EL;
    ss = CurrentSecurityState();
    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_GEN[].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);
base = if ss == SS_Secure then 0 else 4;
if HaltingAllowed() then
    boolean halt;
    if HaveExtendedECDebugEvents() then
        exception_exit = !exception_entry;
        increment = 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 Havev8p8Debug() && 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)
    if HaltingAllowed() && EDESR.SS == '1' then
        // The STATUS code depends on how we arrived at the state where EDESR.SS == 1.
        if HaltingStep_DidNotStep() then
            Halt(DebugHalt_Step_NoSyndrome, is_async);
        elsif HaltingStep_SteppedEX() then
            Halt(DebugHalt_Step_Exclusive, is_async);
        else
            Halt(DebugHalt_Step_Normal, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckOSUnlockCatch

```
// CheckOSUnlockCatch()
// =====
// Called on unlocking the OS Lock to pend an OS Unlock Catch debug event

CheckOSUnlockCatch()
    if ((HaveDoPD() && CTIDEVCTL.OSUCE == '1') ||
        (!HaveDoPD() && 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 Havev8p8Debug() && HaltingAllowed() && EDESR.EC == '1' then
        Halt(DebugHalt_ExceptionCatch, is_async);
```

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
        boolean is_async = TRUE;
        Halt(DebugHalt_OSUnlockCatch, is_async);
```

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
        boolean is_async = TRUE;
        Halt(DebugHalt_ResetCatch, is_async);
```

Library pseudocode for shared/debug/haltingevents/CheckResetCatch

```
// CheckResetCatch()
// =====
// Called after reset

CheckResetCatch()
    if (HaveDoPD() && CTIDEVCTL.RCE == '1') || (!HaveDoPD() && 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() && EDSCR.TDA == '1' && os_lock == '0' then
        Halt(DebugHalt_SoftwareAccess);
```

Library pseudocode for shared/debug/haltingevents/ExternalDebugRequest

```
// ExternalDebugRequest()
// =====

ExternalDebugRequest()
    if HaltingAllowed() then
        boolean is_async = TRUE;
        Halt(DebugHalt_EDBGRQ, is_async);
    // Otherwise the CTI continues to assert the debug request until it is taken.
```

Library pseudocode for shared/debug/haltingevents/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

```
// 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/haltingevents/RunHaltingStep

```
// RunHaltingStep()
// =====

RunHaltingStep(boolean exception_generated, bits(2) exception_target, boolean syscall,
               boolean reset)
    // "exception_generated" is TRUE if the previous instruction generated a synchronous exception
    // or was cancelled by an asynchronous exception.
    //
    // if "exception_generated" is TRUE then "exception_target" is the target of the exception, and
    // "syscall" is TRUE if the exception is a synchronous exception where the preferred return
    // address is the instruction following that which generated the exception.
    //
    // "reset" is TRUE if exiting reset state into the highest EL.

    if reset then assert !Halted();           // Cannot come out of reset halted
    active = EDECR.SS == '1' && !Halted();

    if active && reset then                   // Coming out of reset with EDECR.SS set
        EDESR.SS = '1';
    elsif active && HaltingAllowed() then
        boolean advance;
        if exception_generated && exception_target == EL3 then
            advance = syscall || ExternalSecureInvasiveDebugEnabled();
        else
            advance = TRUE;
        if advance then EDESR.SS = '1';

    return;
```

Library pseudocode for shared/debug/interrupts/ExternalDebugInterruptsDisabled

```
// ExternalDebugInterruptsDisabled()
// =====
// Determine whether EDSCR disables interrupts routed to 'target'.

boolean ExternalDebugInterruptsDisabled(bits(2) target)
    boolean int_dis;
    SecurityState ss = SecurityStateAtEL(target);
    if Havev8p4Debug() then
        if EDSCR.INTdis[0] == '1' then
            case ss of
                when SS_NonSecure int_dis = ExternalInvasiveDebugEnabled();
                when SS_Secure    int_dis = ExternalSecureInvasiveDebugEnabled();
            else
                int_dis = FALSE;
        else
            case target of
                when EL3
                    int_dis = (EDSCR.INTdis == '11' && ExternalSecureInvasiveDebugEnabled());
                when EL2
                    int_dis = (EDSCR.INTdis IN {'1x'} && ExternalInvasiveDebugEnabled());
                when EL1
                    if ss == SS_Secure then
                        int_dis = (EDSCR.INTdis IN {'1x'} && ExternalSecureInvasiveDebugEnabled());
                    else
                        int_dis = (EDSCR.INTdis != '00' && ExternalInvasiveDebugEnabled());
            return int_dis;
```

Library pseudocode for shared/debug/pmu/GetNumEventCounters

```
// GetNumEventCounters()
// =====
// Returns the number of event counters implemented. This is indicated to software at the
// highest Exception level by PMCR.N in AArch32 state, and PMCR_EL0.N in AArch64 state.

integer GetNumEventCounters()
    return integer IMPLEMENTATION_DEFINED "Number of event counters";
```

Library pseudocode for shared/debug/pmu/HasElapsed64Cycles

```
// Returns TRUE if 64 cycles have elapsed between the last count, and FALSE otherwise.
boolean HasElapsed64Cycles();
```

Library pseudocode for shared/debug/pmu/PMUCountValue

```
// PMUCountValue()
// =====
// Implements the PMU threshold function, if implemented.
// Returns the value to increment event counter 'n' by, if the event it is
// configured to count yields the value 'V' on this cycle.

integer PMUCountValue(integer n, integer V)
    if !HavePMUv3TH() then
        return V;

    integer T = UInt(PMEVTYPER_EL0[n].TH);

    case PMEVTYPER_EL0[n].TC of
        when '000' return (if V != T then V else 0); // Disabled or not-equal
        when '001' return (if V != T then 1 else 0); // Not-equal, count
        when '010' return (if V == T then V else 0); // Equals
        when '011' return (if V == T then 1 else 0); // Equals, count
        when '100' return (if V >= T then V else 0); // Greater-than-or-equal
        when '101' return (if V >= T then 1 else 0); // Greater-than-or-equal, count
        when '110' return (if V < T then V else 0); // Less-than
        when '111' return (if V < T then 1 else 0); // Less-than, count
```

Library pseudocode for shared/debug/pmu/PMUCounterMask

```
constant integer CYCLE_COUNTER_ID = 31;

// PMUCounterMask()
// =====
// Return bitmask of accessible PMU counters.

bits(32) PMUCounterMask()
    integer n;
    if UsingAArch32() then
        n = AArch32.GetNumEventCountersAccessible();
    else
        n = AArch64.GetNumEventCountersAccessible();
    return '1' : ZeroExtend(Ones(n), 31);
```

Library pseudocode for shared/debug/pmu/PMUEvent

```
// PMUEvent()  
// =====  
// Generate a PMU event. By default, increment by 1.  
  
PMUEvent(bits(16) event)  
    PMUEvent(event, 1);  
  
// PMUEvent()  
// =====  
// Accumulate a PMU Event.  
  
PMUEvent(bits(16) event, integer increment)  
    integer counters = GetNumEventCounters();  
    if counters != 0 then  
        for idx = 0 to counters - 1  
            PMUEvent(event, increment, idx);  
  
// PMUEvent()  
// =====  
// Accumulate a PMU Event for a specific event counter.  
  
PMUEvent(bits(16) event, integer increment, integer idx)  
    if !HavePMUv3() then  
        return;  
  
    if UsingAArch32() then  
        if PMEVTYPER[idx].evtCount == event then  
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;  
    else  
        if PMEVTYPER_EL0[idx].evtCount == event then  
            PMUEventAccumulator[idx] = PMUEventAccumulator[idx] + increment;
```

Library pseudocode for shared/debug/pmu/integer

```
array integer PMUEventAccumulator[0..30]; // Accumulates PMU events for a cycle
```

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.

pc_sample.valid = ExternalNoninvasiveDebugAllowed() && !Halted();
pc_sample.pc = ThisInstrAddr();
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 !Have16bitVMID() || VTCR_EL2.VS == '0' then
        pc_sample.vmid = ZeroExtend(VTTBR_EL2.VMID<7:0>, 16);
    else
        pc_sample.vmid = VTTBR_EL2.VMID;
    if (HaveVirtHostExt() || HaveV82Debug()) && !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/EDPCSRlo

```
// EDPCSRlo[] (read)
// =====

bits(32) EDPCSRlo[boolean memory_mapped]

if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

// 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 HaveVirtHostExt() && 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 (HaveVirtHostExt() || HaveV82Debug()) && 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());
                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/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
)
PCSample pc_sample;
```

Library pseudocode for shared/debug/samplebasedprofiling/PMPCSR

```
// PMPCSR[] (read)
// =====

bits(32) PMPCSR[boolean memory_mapped]
  if EDPRSR<6:5,0> != '001' then // Check DLK, OSLK and PU bits
    IMPLEMENTATION_DEFINED "generate error response";
    return bits(32) UNKNOWN;

  // The Software lock is OPTIONAL.
  update = !memory_mapped || PMLSR.SLK == '0'; // Software locked: no side-effects

  bits(32) sample;
  if pc_sample.valid then
    sample = pc_sample.pc<31:0>;
    if update then
      PMPCSR<55:32> = (if pc_sample.rw == '0' then Zeros(24) else pc_sample.pc<55:32>);
      PMPCSR.EL = pc_sample.el;
      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);
    else
      sample = Ones(32);
      if update then
        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 sample;
```

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 ==
  if step_enabled && PSTATE.SS == '0' then
    AArch64.SoftwareStepException();
```

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)
  if UsingAArch32() then
    assert N == 32;
  else
    assert N == 64;

  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.
  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';

  return;
```

Library pseudocode for shared/debug/softwarestep/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

```
// 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/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

```
enumeration Exception {Exception_Uncategorized, // Uncategorized or unknown reason
    Exception_WFxTrap, // Trapped WFI or WFE instruction
    Exception_CP15RTTTrap, // Trapped AArch32 MCR or MRC access, coproc=0b1110
    Exception_CP15RRTTrap, // Trapped AArch32 MCRR or MRRC access, coproc=0b1110
    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, // HCPTTR-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 LD
    // 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_SPAAlignment, // 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_BranchTarget, // Branch Target Identification
    Exception_MemCpyMemSet, // Exception from a CPY* or SET* instruction
    Exception_FIQ}; // FIQ interrupt
```

Library pseudocode for shared/exceptions/exceptions/ExceptionRecord

```
type ExceptionRecord is (
    Exception exceptype, // Exception class
    bits(25) syndrome, // Syndrome record
    bits(5) syndrome2, // ST64BV(0) return value register specifier
    bits(64) vaddress, // Virtual fault address
    boolean ipavalid, // Validity of Intermediate Physical fault address
    bit NS, // Intermediate Physical fault address space
    bits(52) 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 = Zeros();
    r.syndrome2 = Zeros();
    r.vaddress = Zeros();
    r.ipavalid = FALSE;
    r.NS = '0';
    r.ipaddress = Zeros();
    r.trappedsyscallinst = FALSE;
    return r;
```

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;

    if level == -1 then
        assert Have52BitIPAAndPASpaceExt();
        case statuscode of
            when Fault_AddressSize          result = '101001';
            when Fault_Translation          result = '101011';
            when Fault_SyncExternalOnWalk   result = '010011';
            when Fault_SyncParityOnWalk     result = '011011'; assert !HaveRASExt();
            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';
        when Fault_Alignment            result = '100001';
        when Fault_Debug                result = '100010';
        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.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);
```

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);
```

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);
```

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

```
// 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/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;
```

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;
```

Library pseudocode for shared/functions/cache/CacheOp

```
enumeration CacheOp {
    CacheOp_Clean,
    CacheOp_Invalidate,
    CacheOp_CleanInvalidate
};
```

Library pseudocode for shared/functions/cache/CacheOpScope

```
enumeration CacheOpScope {
    CacheOpScope_SetWay,
    CacheOpScope_PoU,
    CacheOpScope_PoC,
    CacheOpScope_PoP,
    CacheOpScope_PoDP,
    CacheOpScope_ALLU,
    CacheOpScope_ALLUIS
};
```

Library pseudocode for shared/functions/cache/CachePASpace

```
enumeration CachePASpace {
    CPAS_NonSecure,
    CPAS_SecureNonSecure, // match entries from Secure or Non-Secure PAS
    CPAS_Secure
};
```

Library pseudocode for shared/functions/cache/CacheRecord

```
type CacheRecord is (
  AccType      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      set,              // For SW operations
  integer      way,              // 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 set/way
  // For operations by address, PA space in paddress
  CachePASpace cpas
)
```

Library pseudocode for shared/functions/cache/CacheType

```
enumeration CacheType {
  CacheType_Data,
  CacheType_Tag,
  CacheType_Data_Tag,
  CacheType_Instruction
};
```

Library pseudocode for shared/functions/cache/DCInstNeedsTranslation

```
// DCInstNeedsTranslation()
// =====
// Check whether Data Cache operation needs translation.

boolean DCInstNeedsTranslation(CacheOpScope opscope)
  if CLIDR_EL1.LoC == '000' then
    return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoC is before any

  if CLIDR_EL1.LoUU == '000' && opscope == CacheOpScope_PoU then
    return !boolean IMPLEMENTATION_DEFINED "No fault generated for DC operations if PoU is before any

  return TRUE;
```

Library pseudocode for shared/functions/cache/DecodeSW

```
// DecodeSW()
// =====
// Decode input value into set, way and level for SW instructions.

(integer, integer, integer) DecodeSW(bits(64) regval, CacheType cachetype)
  level = UInt(regval[3:1]);
  (set, way, linesize) = GetCacheInfo(level, cachetype);
  return (set, way, level);
```

Library pseudocode for shared/functions/cache/GetCacheInfo

```
// Returns numsets, associativity & linesize.
(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/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[] - non-assignment form
// =====

bits(size) Elem[bits(N) vector, integer e]
  return Elem[vector, 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;

// Elem[] - assignment form
// =====

Elem[bits(N) &vector, integer e] = bits(size) value
  Elem[vector, 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);

// Extend()
// =====

bits(N) Extend(bits(M) x, boolean unsigned)
    return Extend(x, N, unsigned);
```

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/Int

```
// Int()
// =====

integer Int(bits(N) x, boolean unsigned)
    result = if unsigned then UInt(x) else SInt(x);
    return result;
```

Library pseudocode for shared/functions/common/IsOnes

```
// IsOnes()
// =====

boolean IsOnes(bits(N) x)
    return x == Ones(N);
```

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/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/Ones

```
// Ones()
// =====

bits(N) Ones(integer N)
    return Replicate('1',N);

// Ones()
// =====

bits(N) Ones()
    return Ones(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/Replicate

```
// Replicate()
// =====

bits(N) Replicate(bits(M) x)
    assert N MOD M == 0;
    return Replicate(x, N DIV M);

bits(M*N) Replicate(bits(M) x, integer N);
```

Library pseudocode for shared/functions/common/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

```
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;

// SignExtend()
// =====

bits(N) SignExtend(bits(M) x)
    return SignExtend(x, N);
```

Library pseudocode for shared/functions/common/Split64to32

```
// Split64to32()
// =====

(bits(32), bits(32)) Split64to32(bits(64) value)
    return (value<63:32>, value<31:0>);
```

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;

// ZeroExtend()
// =====

bits(N) ZeroExtend(bits(M) x)
    return ZeroExtend(x, N);
```

Library pseudocode for shared/functions/common/Zeros

```
// Zeros()
// =====

bits(N) Zeros(integer N)
    return Replicate('0',N);

// Zeros()
// =====

bits(N) Zeros()
    return Zeros(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;
  boolean ecv = FALSE;
  if HaveECVExt() then
    ecv = CNTHCTL_EL2.ECV == '1' && SCR_EL3.ECVEn == '1' && EL2Enabled();
  if ecv then
    offset = CNTPOFF_EL2;
  else
    offset = Zeros(64);
  if CNTP_CTL_EL0.ENABLE == '1' then
    imask = CNTP_CTL_EL0.IMASK;
    status = IsTimerConditionMet(offset, CNTP_CVAL_EL0,
                                imask, InterruptID_Cntp);
    CNTP_CTL_EL0.ISTATUS = if status then '1' else '0';
  if ((HaveEL(EL3) || (HaveEL(EL2) && !HaveSecureEL2Ext())) &&
      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) && HaveSecureEL2Ext() && 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(offset, 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;
    status = IsTimerConditionMet(CNTVOFF_EL2, CNTV_CVAL_EL0,
                                imask, InterruptID_CNTV);
    CNTV_CTL_EL0.ISTATUS = if status then '1' else '0';

  if ((HaveVirtHostExt() && (HaveEL(EL3) || !HaveSecureEL2Ext())) &&
      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 ((HaveSecureEL2Ext() && HaveVirtHostExt()) &&
      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/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 HaveCNTSCEExt() && CNTCR.SCEN == '1' then
    PhysicalCount = PhysicalCount + ZeroExtend(CNTSCR);
  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/IsTimerConditionMet

```
// IsTimerConditionMet()
// =====

boolean IsTimerConditionMet(bits(64) offset, bits(64) compare_value,
                           bits(1) imask, InterruptID intid)
  boolean conditon_met;
  signal level;
  condition_met = (UInt(PhysicalCountInt() - offset) -
                  UInt(compare_value)) >= 0;
  level = if condition_met && imask == '0' then HIGH else LOW;
  SetInterruptRequestLevel(intid, level);
  return condition_met;
```

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;
  if CNTHCTL_EL2.EVNTEN == '1' then
    n = UInt(CNTHCTL_EL2.EVNTI);
    if HaveECVExt() && CNTHCTL_EL2.EVNTIS == '1' then
      n = n + 8;
    boolean ecv = FALSE;
    if HaveECVExt() then
      ecv = (EL2Enabled() && CNTHCTL_EL2.ECV == '1' &&
        SCR_EL3.ECVEn == '1');
      offset = if ecv then CNTPOFF_EL2 else 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;
  if (!(HaveVirtHostExt() && HCR_EL2.<E2H,TGE> == '11') &&
    CNTKCTL_EL1.EVNTEN == '1') then
    n = UInt(CNTKCTL_EL1.EVNTI);
    if HaveECVExt() && CNTKCTL_EL1.EVNTIS == '1' then
      n = n + 8;
    if HaveEL(EL2) && (!EL2Enabled() || HCR_EL2.<E2H,TGE> != '11') then
      offset = CNTVOFF_EL2;
    else
      offset = 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/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/HaveCRCExt

```
// HaveCRCExt()
// =====

boolean HaveCRCExt()
    return HasArchVersion\(ARMv8p1\) || boolean IMPLEMENTATION_DEFINED "Have CRC extension";
```

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)
    bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
    bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
    bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
    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
  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*/ 0x619953833cbbefbc8b0f52aae4d3be0a0<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)
  bits(4*8) in0 = op< 96+:8> : op< 64+:8> : op< 32+:8> : op<  0+:8>;
  bits(4*8) in1 = op<104+:8> : op< 72+:8> : op< 40+:8> : op<  8+:8>;
  bits(4*8) in2 = op<112+:8> : op< 80+:8> : op< 48+:8> : op< 16+:8>;
  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
    out1<c*8+:8> =          in0<c*8+:8> EOR FFmul02(in1<c*8+:8>) EOR FFmul03(in2<c*8+:8>) EOR
    out2<c*8+:8> =          in0<c*8+:8> EOR          in1<c*8+:8> EOR FFmul02(in2<c*8+:8>) EOR FFmul03(in3<c*8+:8>)
    out3<c*8+:8> = FFmul03(in0<c*8+:8>) EOR          in1<c*8+:8> EOR          in2<c*8+:8> EOR FFmul02(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/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
    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*/ 0xa89f3c507f02f94585334d43fbaafd0<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(op<i*8+:8>)*8+:8>;
    return out;
```

Library pseudocode for shared/functions/crypto/FFmul02

```
// FFmul02()
// =====

bits(8) FFmul02(bits(8) b)
  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*/ 0xFEFCFAF8F6F4F2F0EEEECEAE8E6E4E2E0<127:0> :
    /*6*/ 0xDEDCDAD8D6D4D2D0CECCAC8C6C4C2C0<127:0> :
    /*5*/ 0xBEBCBAB8B6B4B2B0AEACAAA8A6A4A2A0<127:0> :
    /*4*/ 0x9E9C9A98969492908E8C8A8886848280<127:0> :
    /*3*/ 0x7E7C7A78767472706E6C6A6866646260<127:0> :
    /*2*/ 0x5E5C5A58565452504E4C4A4846444240<127:0> :
    /*1*/ 0x3E3C3A38363432302E2C2A2826242220<127:0> :
    /*0*/ 0x1E1C1A18161412100E0C0A0806040200<127:0>
  );
  return FFmul_02<UInt(b)*8+:8>;
```

Library pseudocode for shared/functions/crypto/FFmul03

```
// FFmul03()
// =====

bits(8) FFmul03(bits(8) b)
  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)
  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)
  bits(256*8) FFmul_0B = (
    /*      F E D C B A 9 8 7 6 5 4 3 2 1 0      */
    /*F*/ 0xA3A8B5BE8F849992FBF0EDE6D7DCC1CA<127:0> :
    /*E*/ 0x1318050E3F3429224B405D56676C717A<127:0> :
    /*D*/ 0xD8D3CEC5F4FFE2E9808B969DACA7BAB1<127:0> :
    /*C*/ 0x68637E75444F5259303B262D1C170A01<127:0> :
    /*B*/ 0x555E434879726F640D061B10212A373C<127:0> :
    /*A*/ 0xE5EEF3F8C9C2DFD4BDB6ABA0919A878C<127:0> :
    /*9*/ 0x2E2538330209141F767D606B5A514C47<127:0> :
    /*8*/ 0x9E958883B2B9A4AFC6CDD0DBEAE1FCF7<127:0> :
    /*7*/ 0x545F424978736E650C071A11202B363D<127:0> :
    /*6*/ 0xE4EFF2F9C8C3DED5BCB7AAA1909B868D<127:0> :
    /*5*/ 0x2F2439320308151E777C616A5B504D46<127:0> :
    /*4*/ 0x9F948982B3B8A5AEC7CCD1DAEBE0FDF6<127:0> :
    /*3*/ 0xA2A9B4BF8E859893FAF1ECE7D6DDC0CB<127:0> :
    /*2*/ 0x1219040F3E3528234A415C57666D707B<127:0> :
    /*1*/ 0xD9D2CFC4F5FEE3E8818A979CADA6BBB0<127:0> :
    /*0*/ 0x69627F74454E5358313A272C1D160B00<127:0>
  );
  return FFmul_0B<UInt>(b)*8+:8;
```

Library pseudocode for shared/functions/crypto/FFmul0D

```
// FFmul0D()
// =====

bits(8) FFmul0D(bits(8) b)
  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*/ 0xFAF7E0EDCEC3D4D9929F8885A6ABBCEB1<127:0> :
    /*A*/ 0x2A27303D1E130409424F5855767B6C61<127:0> :
    /*9*/ 0x414C5B5675786F622924333E1D10070A<127:0> :
    /*8*/ 0x919C8B86A5A8BFB2F9F4E3EECDC0D7DA<127:0> :
    /*7*/ 0x4D40575A7974636E25283F32111C0B06<127:0> :
    /*6*/ 0x9D90878AA9A4B3BEF5F8EFE2C1CCDBD6<127:0> :
    /*5*/ 0xF6FBECE1C2CFD8D59E938489AAA7B0BD<127:0> :
    /*4*/ 0x262B3C31121F08054E4354597A77606D<127:0> :
    /*3*/ 0x202D3A3714190E034845525F7C71666B<127:0> :
    /*2*/ 0xF0FDEAE7C4C9DED39895828FACA1B6BB<127:0> :
    /*1*/ 0x9B96818CAFA2B5B8F3FEE9E4C7CADD0<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)
  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*/ 0xFBFB5E7E9C3CDDFD18B859799B3BDFAFA1<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/HaveAESExt

```
// HaveAESExt()
// =====
// TRUE if AES cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveAESExt()
  return boolean IMPLEMENTATION_DEFINED "Has AES Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveBit128PMULLExt

```
// HaveBit128PMULLExt()
// =====
// TRUE if 128 bit form of PMULL instructions support is implemented,
// FALSE otherwise.

boolean HaveBit128PMULLExt()
    return boolean IMPLEMENTATION_DEFINED "Has 128-bit form of PMULL instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA1Ext

```
// HaveSHA1Ext()
// =====
// TRUE if SHA1 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA1Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA1 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA256Ext

```
// HaveSHA256Ext()
// =====
// TRUE if SHA256 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA256Ext()
    return boolean IMPLEMENTATION_DEFINED "Has SHA256 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA3Ext

```
// HaveSHA3Ext()
// =====
// TRUE if SHA3 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA3Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSHA512Ext

```
// HaveSHA512Ext()
// =====
// TRUE if SHA512 cryptographic instructions support is implemented,
// and when SHA1 and SHA2 basic cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSHA512Ext()
    if !HasArchVersion\(ARMv8p2\) || !(HaveSHA1Ext\(\) && HaveSHA256Ext\(\)) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has SHA512 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM3Ext

```
// HaveSM3Ext()
// =====
// TRUE if SM3 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM3Ext()
  if !HasArchVersion\(ARMv8p2\) then
    return FALSE;
  return boolean IMPLEMENTATION_DEFINED "Has SM3 Crypto instructions";
```

Library pseudocode for shared/functions/crypto/HaveSM4Ext

```
// HaveSM4Ext()
// =====
// TRUE if SM4 cryptographic instructions support is implemented,
// FALSE otherwise.

boolean HaveSM4Ext()
  if !HasArchVersion\(ARMv8p2\) then
    return FALSE;
  return boolean IMPLEMENTATION_DEFINED "Has SM4 Crypto instructions";
```

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;
    bits(2048) sboxstring = 0xd690e9fecce13db716b614c228fb2c052b679a762abe04c3aa441326498606999c4250f491e...

    sboxout = sboxstring<(255-UInt(sboxin))*8+7:(255-UInt(sboxin))*8>;
    return sboxout;
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveByAddress

```
// Clear the global Exclusives monitors for all PEs EXCEPT processorid if they
// record any part of the physical address region of size bytes starting at paddress.
// It is IMPLEMENTATION DEFINED whether the global Exclusives monitor for processorid
// is also cleared if it records any part of the address region.
ClearExclusiveByAddress(FullAddress paddress, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveLocal

```
// Clear the local Exclusives monitor for the specified processorid.
ClearExclusiveLocal(integer processorid);
```

Library pseudocode for shared/functions/exclusive/ClearExclusiveMonitors

```
// ClearExclusiveMonitors()
// =====
// Clear the local Exclusives monitor for the executing PE.

ClearExclusiveMonitors()
    ClearExclusiveLocal(ProcessorID());
```

Library pseudocode for shared/functions/exclusive/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

```
// 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 address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/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 address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveGlobal

```
// Record the physical address region of size bytes starting at address in
// the global Exclusives monitor for processorid.
MarkExclusiveGlobal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/MarkExclusiveLocal

```
// Record the physical address region of size bytes starting at address in
// the local Exclusives monitor for processorid.
MarkExclusiveLocal(FullAddress address, integer processorid, integer size);
```

Library pseudocode for shared/functions/exclusive/ProcessorID

```
// Return the ID of the currently executing PE.
integer ProcessorID();
```

Library pseudocode for shared/functions/extension/AArch32.HaveHPDExt

```
// AArch32.HaveHPDExt()
// =====

boolean AArch32.HaveHPDExt()
    return (HasArchVersion(ARMv8p2) &&
        boolean IMPLEMENTATION_DEFINED "Has AArch32 hierarchical permission disables");
```

Library pseudocode for shared/functions/extension/AArch64.HaveHPDExt

```
// AArch64.HaveHPDExt()
// =====

boolean AArch64.HaveHPDExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Have16bitVMID

```
// Have16bitVMID()
// =====
// Returns TRUE if EL2 and support for a 16-bit VMID are implemented.

boolean Have16bitVMID()
    return (HasArchVersion(ARMv8p1) && HaveEL(EL2) &&
            boolean IMPLEMENTATION_DEFINED "Has 16-bit VMID");
```

Library pseudocode for shared/functions/extension/Have52BitIPAAndPASpaceExt

```
// Have52BitIPAAndPASpaceExt()
// =====
// Returns TRUE if 52-bit IPA and PA extension support
// is implemented, and FALSE otherwise.

boolean Have52BitIPAAndPASpaceExt()
    return (HasArchVersion(ARMv8p7) &&
            boolean IMPLEMENTATION_DEFINED "Has 52-bit IPA and PA support" &&
            Have52BitVAExt() && Have52BitPAExt());
```

Library pseudocode for shared/functions/extension/Have52BitPAExt

```
// Have52BitPAExt()
// =====
// Returns TRUE if Large Physical Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitPAExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has large 52-bit PA/IPA support");
```

Library pseudocode for shared/functions/extension/Have52BitVAExt

```
// Have52BitVAExt()
// =====
// Returns TRUE if Large Virtual Address extension
// support is implemented and FALSE otherwise.

boolean Have52BitVAExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has large 52-bit VA support");
```

Library pseudocode for shared/functions/extension/HaveAArch32BF16Ext

```
// HaveAArch32BF16Ext()
// =====
// Returns TRUE if AArch32 BFloat16 instruction support is implemented, and FALSE otherwise.

boolean HaveAArch32BF16Ext()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 BFloat16 extension");
```

Library pseudocode for shared/functions/extension/HaveAArch32Int8MatMulExt

```
// HaveAArch32Int8MatMulExt()
// =====
// Returns TRUE if AArch32 8-bit integer matrix multiply instruction support
// is implemented, and FALSE otherwise.

boolean HaveAArch32Int8MatMulExt()
    return (HasArchVersion(ARMv8p2) &&
            boolean IMPLEMENTATION_DEFINED "Has AArch32 Int8 Mat Mul extension");
```

Library pseudocode for shared/functions/extension/HaveAccessFlagUpdateExt

```
// HaveAccessFlagUpdateExt()  
// =====  
  
boolean HaveAccessFlagUpdateExt()  
    return HasArchVersion\(ARMv8p1\);
```

Library pseudocode for shared/functions/extension/HaveAltFP

```
// HaveAltFP()  
// =====  
// Returns TRUE if alternative Floating-point extension support  
// is implemented, and FALSE otherwise.  
  
boolean HaveAltFP()  
    return HasArchVersion\(ARMv8p7\);
```

Library pseudocode for shared/functions/extension/HaveAtomicExt

```
// HaveAtomicExt()  
// =====  
  
boolean HaveAtomicExt()  
    return HasArchVersion\(ARMv8p1\);
```

Library pseudocode for shared/functions/extension/HaveBF16Ext

```
// HaveBF16Ext()  
// =====  
// Returns TRUE if AArch64 BFloat16 instruction support is implemented, and FALSE otherwise.  
  
boolean HaveBF16Ext()  
    return (HasArchVersion\(ARMv8p6\) ||  
           (HasArchVersion\(ARMv8p2\) &&  
            boolean IMPLEMENTATION_DEFINED "Has AArch64 BFloat16 extension"));
```

Library pseudocode for shared/functions/extension/HaveBTIExt

```
// HaveBTIExt()  
// =====  
// Returns TRUE if support for Branch Target Identification is implemented.  
  
boolean HaveBTIExt()  
    return HasArchVersion\(ARMv8p5\);
```

Library pseudocode for shared/functions/extension/HaveBlockBBM

```
// HaveBlockBBM()  
// =====  
// Returns TRUE if support for changing block size without requiring  
// break-before-make is implemented.  
  
boolean HaveBlockBBM()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveCNTSCExt

```
// HaveCNTSCExt()
// =====
// Returns TRUE if the Generic Counter Scaling is implemented, and FALSE
// otherwise.

boolean HaveCNTSCExt()
    return (HasArchVersion(ARMv8p4) &&
            boolean IMPLEMENTATION_DEFINED "Has Generic Counter Scaling support");
```

Library pseudocode for shared/functions/extension/HaveCommonNotPrivateTransExt

```
// HaveCommonNotPrivateTransExt()
// =====

boolean HaveCommonNotPrivateTransExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveDGHExt

```
// HaveDGHExt()
// =====
// Returns TRUE if Data Gathering Hint instruction support is implemented, and
// FALSE otherwise.

boolean HaveDGHExt()
    return boolean IMPLEMENTATION_DEFINED "Has AArch64 DGH extension";
```

Library pseudocode for shared/functions/extension/HaveDITExt

```
// HaveDITExt()
// =====

boolean HaveDITExt()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveDOTPExt

```
// HaveDOTPExt()
// =====
// Returns TRUE if Dot Product feature support is implemented, and FALSE otherwise.

boolean HaveDOTPExt()
    return (HasArchVersion(ARMv8p4) ||
            (HasArchVersion(ARMv8p2) &&
             boolean IMPLEMENTATION_DEFINED "Has Dot Product extension"));
```

Library pseudocode for shared/functions/extension/HaveDirtyBitModifierExt

```
// HaveDirtyBitModifierExt()
// =====

boolean HaveDirtyBitModifierExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HaveDoPD

```
// HaveDoPD()
// =====
// Returns TRUE if Debug Over Power Down extension
// support is implemented and FALSE otherwise.

boolean HaveDoPD()
    return HasArchVersion(ARMv8p2) && boolean IMPLEMENTATION_DEFINED "Has DoPD extension";
```

Library pseudocode for shared/functions/extension/HaveDoubleFaultExt

```
// HaveDoubleFaultExt()
// =====

boolean HaveDoubleFaultExt()
    return (HasArchVersion(ARMv8p4) && HaveEL(EL3) && !ELUsingAArch32(EL3) && HaveIESB());
```

Library pseudocode for shared/functions/extension/HaveDoubleLock

```
// HaveDoubleLock()
// =====
// Returns TRUE if support for the OS Double Lock is implemented.

boolean HaveDoubleLock()
    return (!HasArchVersion(ARMv8p4) ||
        boolean IMPLEMENTATION_DEFINED "OS Double Lock is implemented");
```

Library pseudocode for shared/functions/extension/HaveE0PDEExt

```
// HaveE0PDEExt()
// =====
// Returns TRUE if support for constant fault times for unprivileged accesses
// to the memory map is implemented.

boolean HaveE0PDEExt()
    return HasArchVersion(ARMv8p5);
```

Library pseudocode for shared/functions/extension/HaveECVExt

```
// HaveECVExt()
// =====
// Returns TRUE if Enhanced Counter Virtualization extension
// support is implemented, and FALSE otherwise.

boolean HaveECVExt()
    return HasArchVersion(ARMv8p6);
```

Library pseudocode for shared/functions/extension/HaveEMPAMExt

```
// HaveEMPAMExt()
// =====
// Returns TRUE if Enhanced MPAM is implemented, and FALSE otherwise.

boolean HaveEMPAMExt()
    return (HasArchVersion(ARMv8p6) &&
        HaveMPAMExt() &&
        boolean IMPLEMENTATION_DEFINED "Has enhanced MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveExtendedCacheSets

```
// HaveExtendedCacheSets()
// =====

boolean HaveExtendedCacheSets()
    return HasArchVersion(ARMv8p3);
```

Library pseudocode for shared/functions/extension/HaveExtendedECDebugEvents

```
// HaveExtendedECDebugEvents()
// =====

boolean HaveExtendedECDebugEvents()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveExtendedExecuteNeverExt

```
// HaveExtendedExecuteNeverExt()
// =====

boolean HaveExtendedExecuteNeverExt()
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveFCADDExt

```
// HaveFCADDExt()
// =====

boolean HaveFCADDExt()
    return HasArchVersion\(ARMv8p3\);
```

Library pseudocode for shared/functions/extension/HaveFGTExt

```
// HaveFGTExt()
// =====
// Returns TRUE if Fine Grained Trap is implemented, and FALSE otherwise.

boolean HaveFGTExt()
    return HasArchVersion\(ARMv8p6\);
```

Library pseudocode for shared/functions/extension/HaveFJCVTZSExt

```
// HaveFJCVTZSExt()
// =====

boolean HaveFJCVTZSExt()
    return HasArchVersion\(ARMv8p3\);
```

Library pseudocode for shared/functions/extension/HaveFP16MulNoRoundingToFP32Ext

```
// HaveFP16MulNoRoundingToFP32Ext()
// =====
// Returns TRUE if has FP16 multiply with no intermediate rounding accumulate
// to FP32 instructions, and FALSE otherwise

boolean HaveFP16MulNoRoundingToFP32Ext()
    if !HaveFP16Ext\(\) then return FALSE;
    if HasArchVersion\(ARMv8p4\) then return TRUE;
    return (HasArchVersion\(ARMv8p2\) &&
        boolean IMPLEMENTATION_DEFINED "Has accumulate FP16 product into FP32 extension");
```

Library pseudocode for shared/functions/extension/HaveFeatCMOW

```
// HaveFeatCMOW()
// =====
// Returns TRUE if the SCTLR_EL1.CMOW bit is implemented and the SCTLR_EL2.CMOW and
// HCRX_EL2.CMOW bits are implemented if EL2 is implemented.

boolean HaveFeatCMOW()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/HaveFeatHBC

```
// HaveFeatHBC()
// =====
// Returns TRUE if the BC instruction is implemented, and FALSE otherwise.

boolean HaveFeatHBC()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/HaveFeatHCX

```
// HaveFeatHCX()
// =====
// Returns TRUE if HCRX_EL2 Trap Control register is implemented,
// and FALSE otherwise.

boolean HaveFeatHCX()
    return HasArchVersion\(ARMv8p7\);
```

Library pseudocode for shared/functions/extension/HaveFeatHPMN0

```
// HaveFeatHPMN0()
// =====
// Returns TRUE if HDCR.HPMN or MDCR_EL2.HPMN is permitted to be 0 without
// generating UNPREDICTABLE behavior, and FALSE otherwise.

boolean HaveFeatHPMN0()
    return HasArchVersion\(ARMv8p8\) && HavePMUv3\(\) && HaveFGTExt\(\) && HaveEL\(EL2\);
```

Library pseudocode for shared/functions/extension/HaveFeatLS64

```
// HaveFeatLS64()
// =====
// Returns TRUE if the LD64B, ST64B instructions are
// supported, and FALSE otherwise.

boolean HaveFeatLS64()
    return (HasArchVersion\(ARMv8p7\) &&
        boolean IMPLEMENTATION_DEFINED "Has Load Store 64-Byte instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_ACCDATA

```
// HaveFeatLS64_ACCDATA()
// =====
// Returns TRUE if the ST64BV0 instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_ACCDATA()
    return (HasArchVersion\(ARMv8p7\) && HaveFeatLS64\_V\(\) &&
        boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte EL0 with return instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatLS64_V

```
// HaveFeatLS64_V()
// =====
// Returns TRUE if the ST64BV instruction is
// supported, and FALSE otherwise.

boolean HaveFeatLS64_V()
    return (HasArchVersion\(ARMv8p7\) && HaveFeatLS64\(\) &&
        boolean IMPLEMENTATION_DEFINED "Has Store 64-Byte with return instruction support");
```

Library pseudocode for shared/functions/extension/HaveFeatMOPS

```
// HaveFeatMOPS()
// =====
// Returns TRUE if the CPY* and SET* instructions are supported, and FALSE otherwise.

boolean HaveFeatMOPS()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/HaveFeatNMI

```
// HaveFeatNMI()
// =====
// Returns TRUE if the Non-Maskable Interrupt extension is
// implemented, and FALSE otherwise.

boolean HaveFeatNMI()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/HaveFeatRPRES

```
// HaveFeatRPRES()
// =====
// Returns TRUE if reciprocal estimate implements 12-bit precision
// when FPCR.AH=1, and FALSE otherwise.

boolean HaveFeatRPRES()
    return (HasArchVersion\(ARMv8p7\) &&
        (boolean IMPLEMENTATION_DEFINED "Has increased Reciprocal Estimate and Square Root Estimate precision"
            HaveAltFP\(\));
```

Library pseudocode for shared/functions/extension/HaveFeatTIDCP1

```
// HaveFeatTIDCP1()
// =====
// Returns TRUE if the SCTLR_EL1.TIDCP bit is implemented and the SCTLR_EL2.TIDCP bit
// is implemented if EL2 is implemented.

boolean HaveFeatTIDCP1()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/HaveFeatWFxT

```
// HaveFeatWFxT()
// =====
// Returns TRUE if WFET and WFIT instruction support is implemented,
// and FALSE otherwise.

boolean HaveFeatWFxT()
    return HasArchVersion\(ARMv8p7\);
```

Library pseudocode for shared/functions/extension/HaveFeatWFxT2

```
// HaveFeatWFxT2()
// =====
// Returns TRUE if the register number is reported in the ESR_ELx
// on exceptions to WFIT and WFET.

boolean HaveFeatWFxT2()
    return HaveFeatWFxT\(\) && boolean IMPLEMENTATION_DEFINED "Has feature WFxT2";
```

Library pseudocode for shared/functions/extension/HaveFeatXS

```
// HaveFeatXS()
// =====
// Returns TRUE if XS attribute and the TLBI and DSB instructions with nXS qualifier
// are supported, and FALSE otherwise.

boolean HaveFeatXS()
    return HasArchVersion\(ARMv8p7\);
```

Library pseudocode for shared/functions/extension/HaveFlagFormatExt

```
// HaveFlagFormatExt()  
// =====  
// Returns TRUE if flag format conversion instructions implemented.  
  
boolean HaveFlagFormatExt()  
    return HasArchVersion\(ARMv8p5\);
```

Library pseudocode for shared/functions/extension/HaveFlagManipulateExt

```
// HaveFlagManipulateExt()  
// =====  
// Returns TRUE if flag manipulate instructions are implemented.  
  
boolean HaveFlagManipulateExt()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveFrintExt

```
// HaveFrintExt()  
// =====  
// Returns TRUE if FRINT instructions are implemented.  
  
boolean HaveFrintExt()  
    return HasArchVersion\(ARMv8p5\);
```

Library pseudocode for shared/functions/extension/HaveHPMDExt

```
// HaveHPMDExt()  
// =====  
  
boolean HaveHPMDExt()  
    return HavePMUv3p1\(\);
```

Library pseudocode for shared/functions/extension/HaveIDSExt

```
// HaveIDSExt()  
// =====  
// Returns TRUE if ID register handling feature is implemented.  
  
boolean HaveIDSExt()  
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveIESB

```
// HaveIESB()  
// =====  
  
boolean HaveIESB()  
    return (HaveRASExt\(\) &&  
           boolean IMPLEMENTATION_DEFINED "Has Implicit Error Synchronization Barrier");
```

Library pseudocode for shared/functions/extension/HaveInt8MatMulExt

```
// HaveInt8MatMulExt()  
// =====  
// Returns TRUE if AArch64 8-bit integer matrix multiply instruction support  
// implemented, and FALSE otherwise.  
  
boolean HaveInt8MatMulExt()  
    return (HasArchVersion\(ARMv8p6\) ||  
           (HasArchVersion\(ARMv8p2\) &&  
            boolean IMPLEMENTATION_DEFINED "Has AArch64 Int8 Mat Mul extension"));
```

Library pseudocode for shared/functions/extension/HaveLSE2Ext

```
// HaveLSE2Ext()
// =====
// Returns TRUE if LSE2 is implemented, and FALSE otherwise.

boolean HaveLSE2Ext()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/HaveMPAMExt

```
// HaveMPAMExt()
// =====
// Returns TRUE if MPAM is implemented, and FALSE otherwise.

boolean HaveMPAMExt()
    return (HasArchVersion\(ARMv8p2\) &&
        boolean IMPLEMENTATION_DEFINED "Has MPAM extension");
```

Library pseudocode for shared/functions/extension/HaveMTE2Ext

```
// HaveMTE2Ext()
// =====
// Returns TRUE if MTE support is beyond EL0, and FALSE otherwise.

boolean HaveMTE2Ext()
    if !HasArchVersion\(ARMv8p5\) then
        return FALSE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE2 extension";
```

Library pseudocode for shared/functions/extension/HaveMTE3Ext

```
// HaveMTE3Ext()
// =====
// Returns TRUE if MTE Asymmetric Fault Handling support is
// implemented, and FALSE otherwise.

boolean HaveMTE3Ext()
    return ((HasArchVersion\(ARMv8p7\) && HaveMTE2Ext\(\)) || (HasArchVersion\(ARMv8p5\) &&
        boolean IMPLEMENTATION_DEFINED "Has MTE3 extension"));
```

Library pseudocode for shared/functions/extension/HaveMTEExt

```
// HaveMTEExt()
// =====
// Returns TRUE if MTE implemented, and FALSE otherwise.

boolean HaveMTEExt()
    if !HasArchVersion\(ARMv8p5\) then
        return FALSE;
    if HaveMTE2Ext\(\) then
        return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Has MTE extension";
```

Library pseudocode for shared/functions/extension/HaveNV2Ext

```
// HaveNV2Ext()
// =====
// Returns TRUE if Enhanced Nested Virtualization is implemented.

boolean HaveNV2Ext()
    return (HasArchVersion\(ARMv8p4\) && HaveNVExt\(\)
        && boolean IMPLEMENTATION_DEFINED "Has support for Enhanced Nested Virtualization");
```

Library pseudocode for shared/functions/extension/HaveNVExt

```
// HaveNVExt()  
// =====  
// Returns TRUE if Nested Virtualization is implemented.  
  
boolean HaveNVExt()  
    return (HasArchVersion(ARMv8p3) &&  
            boolean IMPLEMENTATION_DEFINED "Has Nested Virtualization");
```

Library pseudocode for shared/functions/extension/HaveNoSecurePMUDisableOverride

```
// HaveNoSecurePMUDisableOverride()  
// =====  
  
boolean HaveNoSecurePMUDisableOverride()  
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveNoninvasiveDebugAuth

```
// HaveNoninvasiveDebugAuth()  
// =====  
// Returns TRUE if the Non-invasive debug controls are implemented.  
  
boolean HaveNoninvasiveDebugAuth()  
    return !HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HavePAN3Ext

```
// HavePAN3Ext()  
// =====  
// Returns TRUE if SCTL_EL1.EPAN and SCTL_EL2.EPAN support is implemented,  
// and FALSE otherwise.  
  
boolean HavePAN3Ext()  
    return HasArchVersion(ARMv8p7) || (HasArchVersion(ARMv8p1) &&  
            boolean IMPLEMENTATION_DEFINED "Has PAN3 extension");
```

Library pseudocode for shared/functions/extension/HavePANExt

```
// HavePANExt()  
// =====  
  
boolean HavePANExt()  
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/HavePMUv3

```
// HavePMUv3()  
// =====  
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3()  
    return boolean IMPLEMENTATION_DEFINED "Has Performance Monitors extension";
```

Library pseudocode for shared/functions/extension/HavePMUv3TH

```
// HavePMUv3TH()  
// =====  
// Returns TRUE if the PMUv3 threshold extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3TH()  
    return (HasArchVersion(ARMv8p8) && HavePMUv3()) &&  
            boolean IMPLEMENTATION_DEFINED "Has PMUv3 threshold extension");
```

Library pseudocode for shared/functions/extension/HavePMUv3p1

```
// HavePMUv3p1()  
// =====  
// Returns TRUE if the Performance Monitors extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p1()  
    return HasArchVersion\(ARMv8p1\) && HavePMUv3\(\);
```

Library pseudocode for shared/functions/extension/HavePMUv3p4

```
// HavePMUv3p4()  
// =====  
// Returns TRUE if the PMUv3.4 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p4()  
    return HasArchVersion\(ARMv8p4\) && HavePMUv3\(\);
```

Library pseudocode for shared/functions/extension/HavePMUv3p5

```
// HavePMUv3p5()  
// =====  
// Returns TRUE if the PMUv3.5 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p5()  
    return HasArchVersion\(ARMv8p5\) && HavePMUv3\(\);
```

Library pseudocode for shared/functions/extension/HavePMUv3p7

```
// HavePMUv3p7()  
// =====  
// Returns TRUE if the PMUv3.7 extension is implemented, and FALSE otherwise.  
  
boolean HavePMUv3p7()  
    return HasArchVersion\(ARMv8p7\) && HavePMUv3\(\);
```

Library pseudocode for shared/functions/extension/HavePageBasedHardwareAttributes

```
// HavePageBasedHardwareAttributes()  
// =====  
  
boolean HavePageBasedHardwareAttributes()  
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HavePrivATExt

```
// HavePrivATExt()  
// =====  
  
boolean HavePrivATExt()  
    return HasArchVersion\(ARMv8p2\);
```

Library pseudocode for shared/functions/extension/HaveQRDMLAHExt

```
// HaveQRDMLAHExt()  
// =====  
  
boolean HaveQRDMLAHExt()  
    return HasArchVersion\(ARMv8p1\);
```

Library pseudocode for shared/functions/extension/HaveRASExt

```
// HaveRASExt()  
// =====  
  
boolean HaveRASExt()  
    return (HasArchVersion(ARMv8p2) ||  
            boolean IMPLEMENTATION_DEFINED "Has RAS extension");
```

Library pseudocode for shared/functions/extension/HaveRNG

```
// HaveRNG()  
// =====  
// Returns TRUE if Random Number Generator extension  
// support is implemented and FALSE otherwise.  
  
boolean HaveRNG()  
    return HasArchVersion(ARMv8p5) && boolean IMPLEMENTATION_DEFINED "Has RNG extension";
```

Library pseudocode for shared/functions/extension/HaveSBExt

```
// HaveSBExt()  
// =====  
// Returns TRUE if support for SB is implemented, and FALSE otherwise.  
  
boolean HaveSBExt()  
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SB extension";
```

Library pseudocode for shared/functions/extension/HaveSSBSExt

```
// HaveSSBSExt()  
// =====  
// Returns TRUE if support for SSBS is implemented, and FALSE otherwise.  
  
boolean HaveSSBSExt()  
    return HasArchVersion(ARMv8p5) || boolean IMPLEMENTATION_DEFINED "Has SSBS extension";
```

Library pseudocode for shared/functions/extension/HaveSecureEL2Ext

```
// HaveSecureEL2Ext()  
// =====  
// Returns TRUE if Secure EL2 is implemented.  
  
boolean HaveSecureEL2Ext()  
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSecureExtDebugView

```
// HaveSecureExtDebugView()  
// =====  
// Returns TRUE if support for Secure and Non-secure views of debug peripherals  
// is implemented.  
  
boolean HaveSecureExtDebugView()  
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSelfHostedTrace

```
// HaveSelfHostedTrace()  
// =====  
  
boolean HaveSelfHostedTrace()  
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveSmallTranslationTblExt

```
// HaveSmallTranslationTblExt()
// =====
// Returns TRUE if Small Translation Table Support is implemented.

boolean HaveSmallTranslationTableExt()
    return (HasArchVersion(ARMv8p4) &&
        boolean IMPLEMENTATION_DEFINED "Has Small Translation Table extension");
```

Library pseudocode for shared/functions/extension/HaveSoftwareLock

```
// HaveSoftwareLock()
// =====
// Returns TRUE if Software Lock is implemented.

boolean HaveSoftwareLock(Component component)
    if Havev8p4Debug() then
        return FALSE;
    if HaveDoPD() && component != Component_CTI then
        return FALSE;
    case component of
        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/HaveStage2MemAttrControl

```
// HaveStage2MemAttrControl()
// =====
// Returns TRUE if support for Stage2 control of memory types and cacheability
// attributes is implemented.

boolean HaveStage2MemAttrControl()
    return HasArchVersion(ARMv8p4);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfiling

```
// HaveStatisticalProfiling()
// =====
// Returns TRUE if Statistical Profiling Extension is implemented,
// and FALSE otherwise.

boolean HaveStatisticalProfiling()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p1

```
// HaveStatisticalProfilingv1p1()
// =====
// Returns TRUE if the SPEv1p1 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p1()
    return (HasArchVersion(ARMv8p3) &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p1 extension");
```

Library pseudocode for shared/functions/extension/HaveStatisticalProfilingv1p2

```
// HaveStatisticalProfilingv1p2()
// =====
// Returns TRUE if the SPEv1p2 extension is implemented, and FALSE otherwise.

boolean HaveStatisticalProfilingv1p2()
    return (HasArchVersion(ARMv8p7) && HaveStatisticalProfiling() &&
        boolean IMPLEMENTATION_DEFINED "Has SPEv1p2 extension");
```

Library pseudocode for shared/functions/extension/HaveTWEDEExt

```
// HaveTWEDEExt()
// =====
// Returns TRUE if Delayed Trapping of WFE instruction support is implemented,
// and FALSE otherwise.

boolean HaveTWEDEExt()
    return boolean IMPLEMENTATION_DEFINED "Has TWED extension";
```

Library pseudocode for shared/functions/extension/HaveTraceExt

```
// HaveTraceExt()
// =====
// Returns TRUE if Trace functionality as described by the Trace Architecture
// is implemented.

boolean HaveTraceExt()
    return boolean IMPLEMENTATION_DEFINED "Has Trace Architecture functionality";
```

Library pseudocode for shared/functions/extension/HaveTrapLoadStoreMultipleDeviceExt

```
// HaveTrapLoadStoreMultipleDeviceExt()
// =====

boolean HaveTrapLoadStoreMultipleDeviceExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveUAOExt

```
// HaveUAOExt()
// =====

boolean HaveUAOExt()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveV82Debug

```
// HaveV82Debug()
// =====

boolean HaveV82Debug()
    return HasArchVersion(ARMv8p2);
```

Library pseudocode for shared/functions/extension/HaveVirtHostExt

```
// HaveVirtHostExt()
// =====

boolean HaveVirtHostExt()
    return HasArchVersion(ARMv8p1);
```

Library pseudocode for shared/functions/extension/Havev8p4Debug

```
// Havev8p4Debug()
// =====
// Returns TRUE if support for the Debugv8p4 feature is implemented and FALSE otherwise.

boolean Havev8p4Debug()
    return HasArchVersion\(ARMv8p4\);
```

Library pseudocode for shared/functions/extension/Havev8p8Debug

```
// Havev8p8Debug()
// =====
// Returns TRUE if support for the Debugv8p8 feature is implemented and FALSE otherwise.

boolean Havev8p8Debug()
    return HasArchVersion\(ARMv8p8\);
```

Library pseudocode for shared/functions/extension/InsertIESBBeforeException

```
// If SCTL_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);
```

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} ||
            (!HaveRASExt\(\) && memretstatus.statuscode IN {Fault\_SyncParity,
                                                         Fault\_AsyncParity}));

    fault = NoFault\(\);
    fault.statuscode = memretstatus.statuscode;
    fault.write = iswrite;
    fault.extflag = memretstatus.extflag;
    fault.acctype = memretstatus.acctype;
    // It is implementation specific whether External aborts signaled
    // in-band synchronously are taken synchronously or asynchronously
    if (IsExternalSyncAbort(fault) &&
        !IsExternalAbortTakenSynchronously(memretstatus, iswrite, memaddrdesc,
                                           size, accdesc)) then
        if fault.statuscode == Fault\_SyncParity then
            fault.statuscode = Fault\_AsyncParity;
        else
            fault.statuscode = Fault\_AsyncExternal;

    if HaveRASExt\(\) then
        fault.errortype = PEErrorState(memretstatus);
    else
        fault.errortype = bits(2) UNKNOWN;

    if IsExternalSyncAbort(fault) then
        if UsingAArch32\(\) then
            AArch32.Abort(memaddrdesc.vaddress<31:0>, fault);
        else
            AArch64.Abort(memaddrdesc.vaddress, 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) &&
        !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 HaveRASExt() then
        output_fault.errortype = PEErrrorState(memretstatus);
    else
        output_fault.errortype = bits(2) UNKNOWN;
    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

```
// 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/PEErrorState

```
constant bits(2) Sync_UC    = '10'; // Synchronous Uncontainable
constant bits(2) Sync_UER   = '00'; // Synchronous Recoverable
constant bits(2) Sync_UE0   = '11'; // Synchronous Restartable
constant bits(2) ASync_UC   = '00'; // ASynchronous Uncontainable
constant bits(2) ASync_UEU  = '01'; // ASynchronous Unrecoverable
constant bits(2) ASync_UER  = '11'; // ASynchronous Recoverable
constant bits(2) ASync_UE0  = '10'; // ASynchronous Restartable

bits(2) PEErrorState(PhysMemRetStatus memstatus);
```

Library pseudocode for shared/functions/externalaborts/PendSErrorInterrupt

```
// Pend the SError.
PendSErrorInterrupt(FaultRecord fault);
```

Library pseudocode for shared/functions/float/bfloat/BFAdd

```
// BFAdd()
// =====
// Single-precision add following BFloat16 computation behaviors.

bits(32) BFAdd(bits(32) op1, bits(32) op2)

    bits(32) result;

    FPCRTType fpcr = FPCR[];
    (type1,sign1,value1) = BFUnpack(op1);
    (type2,sign2,value2) = BFUnpack(op2);
    if type1 == FPTType\_QNaN || type2 == FPTType\_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);
        if inf1 && inf2 && sign1 == NOT(sign2) then
            result = FPDefaultNaN(fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        else
            result_value = value1 + value2;
            if result_value == 0.0 then
                result = FPZero('0'); // Positive sign when Round to Odd
            else
                result = BFRound(result_value);

    return result;
```

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, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;

    bits(32) prod;

    prod = BFAdd(BFMul(op1_a, op2_a), BFMul(op1_b, op2_b));
    result = BFAdd(addend, prod);

    return result;
```

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)

    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
                bits(16) elt1_a = Elem[op1, 4*i + 2*k + 0, 16];
                bits(16) elt1_b = Elem[op1, 4*i + 2*k + 1, 16];
                bits(16) elt2_a = Elem[op2, 4*j + 2*k + 0, 16];
                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/BFMul

```
// BFMul()
// =====
// BFloat16 widening multiply to single-precision following BFloat16
// computation behaviors.

bits(32) BFMul(bits(16) op1, bits(16) op2)

    bits(32) result;

    FPCRType fpcr = FPCR[];
    (type1, sign1, value1) = BFUnpack(op1);
    (type2, sign2, value2) = BFUnpack(op2);
    if type1 == FPType_QNaN || type2 == FPType_QNaN then
        result = FPDefaultNaN(fpcr);
    else
        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);
        elsif inf1 || inf2 then
            result = FPIfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = BFRound(value1*value2);

    return result;
```

Library pseudocode for shared/functions/float/bfloat/BFMulAdd

```
// BFMulAdd()
// =====
// Used by BFMLALB and BFMLALT instructions.

bits(N) BFMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr_in)
    FPCRTType fpcr = fpcr_in;
    boolean altfp = HaveAltFP() && fpcr.AH == '1'; // When TRUE:
    boolean fpexc = !altfp; // Do not generate 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
    return FPMulAdd(addend, op1, op2, fpcr, fpexc);
```

Library pseudocode for shared/functions/float/bfloat/BFNeg

```
// BFNeg()
// =====

bits(16) BFNeg(bits(16) op)
    return NOT(op<15>) : op<14:0>;
```

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(32) BFRound(real op)

    assert op != 0.0;
    bits(32) result;

    // Format parameters - minimum exponent, numbers of exponent and fraction bits.
    minimum_exp = -126; E = 8; F = 23;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    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;

    // Fixed Flush-to-zero.
    if exponent < minimum_exp then
        return FPZero(sign);

    // 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 then
        int_mant<0> = '1';

    // Deal with overflow and generate result.
    if biased_exp >= 2^E - 1 then
        result = FPInfinity(sign); // Overflows generate appropriately-signed Infinity
    else
        result = sign : biased_exp<30-F:0> : int_mant<F-1:0>;

    return result;
```

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/FPConvertBF

```
// FPConvertBF()
// =====
// Converts a single-precision 'op' to BFloat16 value with using rounding mode of
// Round to Nearest Even when executed from AArch64 state and
// FPCR.AH == '1', otherwise rounding is controlled by FPCR/FPSCR.

bits(16) FPConvertBF(bits(32) op, FPCRTYPE fpcr_in, FPRounding rounding_in)

FPCRTYPE fpcr = fpcr_in;
FPRounding rounding = rounding_in;
bits(32) result; // BF16 value in top 16 bits
boolean altfp = HaveAltFP\(\) && !UsingAArch32\(\) && fpcr.AH == '1';
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'.
(ftype,sign,value) = FPUntpack(op, fpcr, fpexc);

if ftype == FPTYPE\_SNaN || ftype == FPTYPE\_QNaN then
    if fpcr.DN == '1' then
        result = FPDefaultNaN(fpcr);
    else
        result = FPConvertNaN(op);
    if ftype == FPTYPE\_SNaN then
        if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
elseif ftype == FPTYPE\_Infinity then
    result = FPInfinity(sign);
elseif ftype == FPTYPE\_Zero then
    result = FPZero(sign);
else
    result = FPRoundCVBF(value, fpcr, rounding, fpexc);

// Returns correctly rounded BF16 value from top 16 bits
return result<31:16>;

// FPConvertBF()
// =====
// Converts a single-precision operand to BFloat16 value.

bits(16) FPConvertBF(bits(32) op, FPCRTYPE fpcr)
    return FPConvertBF(op, fpcr, FPRoundingMode(fpcr));
```

Library pseudocode for shared/functions/float/bfloat/FPRoundCVBF

```
// FPRoundCVBF()
// =====
// Converts a real number 'op' into a BFloat16 value using the supplied
// rounding mode 'rounding'. The 'fpexc' argument controls the generation of
// floating-point exceptions.

bits(32) FPRoundCVBF(real op, FPCRTYPE fpcr, FPRounding rounding, boolean fpexc)
    boolean isbfloat16 = TRUE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);
```

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, FPCRTYPE fpcr, FPRounding rounding)

    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');
    else
        result = FPRound(real_operand, fpcr, rounding);

    return result;
```

Library pseudocode for shared/functions/float/fpabs/FPAbs

```
// FPAbs()
// =====

bits(N) FPAbs(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTYPE fpcr = FPCR[];
        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/fpadd/FPAdd

```
// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPAdd(op1, op2, fpcr, fpexc);

// FPAdd()
// =====

bits(N) FPAdd(bits(N) op1, bits(N) op2, FPCRTType 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);

    boolean altfmaxfmin = FALSE; // Do not use altfp mode for FMIN, FMAX and variants
    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, 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);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '0') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '1') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == sign2 then
            result = FPZero(sign1);
        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);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

        if fpexc then FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcmpare/FPCompare

```
// FPCompare()
// =====

bits(4) FPCompare(bits(N) op1, bits(N) op2, boolean signal_nans, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);

    bits(4) result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = '0011';
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN || signal_nans then
            FPPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUunpack()
        if value1 == value2 then
            result = '0110';
        elsif value1 < value2 then
            result = '1000';
        else // value1 > value2
            result = '0010';

        FPPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcmpareeq/FPCompareEQ

```
// FPCompareEQ()
// =====

boolean FPCompareEQ(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUunpack(op1, fpcr);
    (type2,sign2,value2) = FPUunpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        if type1 == FPTType\_SNaN || type2 == FPTType\_SNaN then
            FPPProcessException(FPExc\_InvalidOp, fpcr);
    else
        // All non-NaN cases can be evaluated on the values produced by FPUunpack()
        result = (value1 == value2);

        FPPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpcmparege/FPCompareGE

```
// FPCompareGE()
// =====

boolean FPCompareGE(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        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 shared/functions/float/fpcmparegt/FPCompareGT

```
// FPCompareGT()
// =====

boolean FPCompareGT(bits(N) op1, bits(N) op2, FPCRTType fpcr)

    assert N IN {16,32,64};
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    boolean result;
    if type1 IN {FPTType\_SNaN, FPTType\_QNaN} || type2 IN {FPTType\_SNaN, FPTType\_QNaN} then
        result = FALSE;
        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 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, FPCRType fpcr, FPRounding rounding)

    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);
        elsif fpcr.DN == '1' then
            result = FPDefaultNaN(fpcr);
        else
            result = FPConvertNaN(op);
            if fptype == FPTYPE\_SNaN || alt_hp then
                FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTYPE\_Infinity then
        if alt_hp then
            result = sign:Ones(M-1);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        else
            result = FPInfinity(sign);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero(sign);
    else
        result = FPRoundCV(value, fpcr, rounding);

        FPProcessDenorm(fptype, N, fpcr);

    return result;

// FPConvert()
// =====

bits(M) FPConvert(bits(N) op, FPCRType fpcr)
    return FPConvert(op, fpcr, FPRoundingMode(fpcr));
```

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)

    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/fpcrtype/FPCRTType

```
type FPCRTType;
```

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()
    FPCRTType fpcr = FPCR[];
    return FPDefaultNaN(fpcr);

bits(N) FPDefaultNaN(FPCRTType fpcr)

    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);
    bit sign = if HaveAltFP() && !UsingAArch32() then fpcr.AH else '0';

    bits(E) exp = Ones(E);
    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, FPCRTType 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);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || zero2 then
            result = FPInfinity(sign1 EOR sign2);
            if !inf1 then FPProcessException(FPExc\_DivideByZero, fpcr);
        elsif zero1 || inf2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1/value2, fpcr);

        if !zero2 then
            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/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)

    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);
    bits(E) exp = Ones(E);
    bits(F) frac = Zeros(F);

    return sign : exp : frac;
```

Library pseudocode for shared/functions/float/fpmax/FPMax

```
// FPMax()
// =====

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMax(op1, op2, fpcr, altfp);

// 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 behaviour.

bits(N) FPMax(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FType\_Zero && type2 == FType\_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        FType fptype;
        bit sign;
        real value;
        if value1 > value2 then
            (fptype,sign,value) = (type1,sign1,value1);
        else
            (fptype,sign,value) = (type2,sign2,value2);
        if fptype == FType\_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FType\_Zero then
            sign = sign1 AND sign2;          // Use most positive sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpmaxnormal/FPMaxNormal

```
// FPMaxNormal()
// =====

bits(N) FPMaxNormal(bit sign)

    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_in, bits(N) op2_in, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean type2_nan = type2 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as -Infinity.
        if type1 == FPTType\_QNaN && type2 != FPTType\_QNaN then
            op1 = FPInfinity('1');
        elsif type1 != FPTType\_QNaN && type2 == FPTType\_QNaN then
            op2 = FPInfinity('1');

    altfmaxfmin = FALSE; // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMax(op1, op2, fpcr, altfmaxfmin);

    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(FPCRTType fpcr)
    boolean merge = HaveAltFP() && !UsingAArch32() && fpcr.NEP == '1';
    return merge;
```

Library pseudocode for shared/functions/float/fpmin/FPMin

```
// FPMin()
// =====

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr)
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    return FPMin(op1, op2, fpcr, altfp);

// FPMin()
// =====
// Compare two operands and return the smaller operand after rounding. The
// 'fpcr' argument supplies the FPCR control bits and 'altfp' determines
// if the function should use alternative behaviour.

bits(N) FPMin(bits(N) op1, bits(N) op2, FPCRTType fpcr_in, boolean altfp)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    (type1,sign1,value1) = FPUnpack(op1, fpcr);
    (type2,sign2,value2) = FPUnpack(op2, fpcr);

    if (altfp && type1 == FType\_Zero && type2 == FType\_Zero &&
        ((sign1 == '0' && sign2 == '1') || (sign1 == '1' && sign2 == '0'))) then
        return FPZero(sign2);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr, altfp, TRUE);

    if !done then
        FType 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 == FType\_Infinity then
            result = FPInfinity(sign);
        elsif fptype == FType\_Zero then
            sign = sign1 OR sign2; // Use most negative sign
            result = FPZero(sign);
        else
            // The use of FPRound() covers the case where there is a trapped underflow exception
            // for a denormalized number even though the result is exact.
            rounding = FPRoundingMode(fpcr);
            if altfp then // Denormal output is not flushed to zero
                fpcr.FZ = '0';
                fpcr.FZ16 = '0';

            result = FPRound(value, fpcr, rounding, TRUE);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpminnum/FPMinNum

```
// FPMinNum()
// =====

bits(N) FPMinNum(bits(N) op1_in, bits(N) op2_in, FPCRTType fpcr)

    assert N IN {16,32,64};
    bits(N) op1 = op1_in;
    bits(N) op2 = op2_in;
    (type1,-,-) = FPUnpack(op1, fpcr);
    (type2,-,-) = FPUnpack(op2, fpcr);

    boolean type1_nan = type1 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean type2_nan = type2 IN {FPTType\_QNaN, FPTType\_SNaN};
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';

    if !(altfp && type1_nan && type2_nan) then
        // Treat a single quiet-NaN as +Infinity.
        if type1 == FPTType\_QNaN && type2 != FPTType\_QNaN then
            op1 = FPInfinity('0');
        elsif type1 != FPTType\_QNaN && type2 == FPTType\_QNaN then
            op2 = FPInfinity('0');

    altfmaxfmin = FALSE;    // Restrict use of FMAX/FMIN NaN propagation rules
    result = FPMin(op1, op2, fpcr, altfmaxfmin);

    return result;
```

Library pseudocode for shared/functions/float/fpmul/FPMul

```
// FPMul()
// =====

bits(N) FPMul(bits(N) op1, bits(N) op2, FPCRTType 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 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

        FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```



```

// FPMulAdd()
// =====

bits(N) FPMulAdd(bits(N) addend, bits(N) op1, bits(N) op2, FPCRTType fpcr)
    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,
    FPCRTType 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 == FPTType\_Infinity); zero1 = (type1 == FPTType\_Zero);
    inf2 = (type2 == FPTType\_Infinity); zero2 = (type2 == FPTType\_Zero);

    (done,result) = FPProcessNaNs3(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTType\_0NaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTType\_Infinity); zeroA = (typeA == FPTType\_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);
            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');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // 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);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then

```

```
    FPProcessDenorms3(typeA, type1, type2, N, fpcr);  
return result;
```



```

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2, FPCRTType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPMulAddH(addend, op1, op2, fpcr, fpexc);

// FPMulAddH()
// =====
// Calculates addend + op1*op2.

bits(N) FPMulAddH(bits(N) addend, bits(N DIV 2) op1, bits(N DIV 2) op2,
    FPCRTType fpcr, boolean fpexc)

    assert N == 32;
    rounding = FPRoundingMode(fpcr);
    (typeA,signA,valueA) = FPUntpack(addend, fpcr, fpexc);
    (type1,sign1,value1) = FPUntpack(op1, fpcr, fpexc);
    (type2,sign2,value2) = FPUntpack(op2, fpcr, fpexc);
    inf1 = (type1 == FPTType\_Infinity); zero1 = (type1 == FPTType\_Zero);
    inf2 = (type2 == FPTType\_Infinity); zero2 = (type2 == FPTType\_Zero);

    (done,result) = FPProcessNaNs3H(typeA, type1, type2, addend, op1, op2, fpcr, fpexc);

    if !(HaveAltFP() && !UsingAArch32() && fpcr.AH == '1') then
        if typeA == FPTType\_QNaN && ((inf1 && zero2) || (zero1 && inf2)) then
            result = FPDefaultNaN(fpcr);
            if fpexc then FPProcessException(FPExc\_InvalidOp, fpcr);

    if !done then
        infA = (typeA == FPTType\_Infinity); zeroA = (typeA == FPTType\_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);
            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');
        elsif (infA && signA == '1') || (infP && signP == '1') then
            result = FPInfinity('1');

        // Cases where the result is exactly zero and its sign is not determined by the
        // rounding mode are additions of same-signed zeros.
        elsif zeroA && zeroP && signA == signP then
            result = FPZero(signA);

        // 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);
            else
                result = FPRound(result_value, fpcr, rounding, fpexc);

    if !invalidop && fpexc then
        FPProcessDenorm(typeA, N, fpcr);

```

```
return result;
```

Library pseudocode for shared/functions/float/fpmuladdh/FPPProcessNaNs3H

```
// FPPProcessNaNs3H()
// =====

(boolean, bits(N)) FPPProcessNaNs3H(FPType type1, FPType type2, FPType type3,
                                     bits(N) op1, bits(N DIV 2) op2, bits(N DIV 2) op3,
                                     FPCRTYPE fpcr, boolean fpexc)

    assert N IN {32,64};

    bits(N) result;
    FPType type_nan;
    // When TRUE, use alternative NaN propagation rules.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};
    if altfp then
        if (type1 == FPType_SNaN || type2 == FPType_SNaN || type3 == FPType_SNaN) then
            type_nan = FPType_SNaN;
        else
            type_nan = FPType_QNaN;

    boolean done;
    if altfp && op1_nan && op2_nan && op3_nan then // <n> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op2_nan && (op1_nan || op3_nan) then // <n> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op2, fpcr, fpexc));
    elsif altfp && op3_nan && op1_nan then // <m> register NaN selected
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type_nan, op3, fpcr, fpexc));
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type2, op2, fpcr, fpexc));
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPConvertNaN(FPPProcessNaN(type3, op3, fpcr, fpexc));
    else
        done = FALSE; result = Zeros(); // 'Don't care' result
    return (done, result);
```

Library pseudocode for shared/functions/float/fpmulx/FPMuIX

```
// FPMuIX()
// =====

bits(N) FPMuIX(bits(N) op1, bits(N) op2, FPCRTType 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 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if (inf1 && zero2) || (zero1 && inf2) then
            result = FPTwo(sign1 EOR sign2);
        elsif inf1 || inf2 then
            result = FPInfinity(sign1 EOR sign2);
        elsif zero1 || zero2 then
            result = FPZero(sign1 EOR sign2);
        else
            result = FPRound(value1*value2, fpcr);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpneg/FPNeg

```
// FPNeg()
// =====

bits(N) FPNeg(bits(N) op)

    assert N IN {16,32,64};
    if !UsingAArch32() && HaveAltFP() then
        FPCRTType fpcr = FPCR[];
        if fpcr.AH == '1' then
            (fptype, -, -) = FPUnpack(op, fpcr, FALSE);
            if fptype IN {FPTType\_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)

    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, FPCRTYPE fpcr)
  boolean altfp = HaveAltFP() && !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, FPCRTYPE fpcr)
  boolean altfp = HaveAltFP() && !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, FPCRTYPE fpcr)
  boolean altfp = HaveAltFP() && !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, FPCRTYPE fpcr)
  boolean altfp = HaveAltFP() && !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/FPProcessException

```
// FPProcessException()
// =====
//
// The 'fpcr' argument supplies FPCR control bits. Status information is
// updated directly in the FPSR where appropriate.

FPProcessException(FPExc exception, FPCRTYPE fpcr)

integer cumul;
// Determine the cumulative exception bit number
case exception 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' 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
    FPSCR<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, FPCRType fpcr)
    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, FPCRType 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);

    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, FPCRType fpcr)
    boolean altfmaxfmin = FALSE; // Do not use alfp mode for FMIN, FMAX and variants
    boolean fpxc        = TRUE;  // Generate floating-point exceptions
    return FPProcessNaNs(type1, type2, op1, op2, fpcr, altfmaxfmin, fpxc);

// 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 behaviour for FMAX, FMIN and variants. 'fpxc'
// 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,
                                FPCRType fpcr, boolean altfmaxfmin, boolean fpxc)

    assert N IN {16,32,64};
    bit sign2;
    boolean done;
    bits(N) result;
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean any_snan = type1 == FPType_SNaN || type2 == FPType_SNaN;
    FPType type_nan = if any_snan then FPType_SNaN else FPType_QNaN;

    if altfmaxfmin && (op1_nan || op2_nan) then
        FPProcessException(FPExc_InvalidOp, fpcr);
        done = TRUE; sign2 = op2<N-1>;
        result = if type2 == FPType_Zero then FPZero(sign2) else op2;
    elsif altfp && op1_nan && op2_nan then
        // <n> register NaN selected
        done = TRUE; result = FPProcessNaN(type_nan, op1, fpcr, fpxc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpxc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpxc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type1, op1, fpcr, fpxc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPProcessNaN(type2, op2, fpcr, fpxc);
    else
        done = FALSE; result = Zeros(); // 'Don't care' result

    return (done, result);
```

Library pseudocode for shared/functions/float/fpprocessnans3/FPPProcessNaNs3

```
// FPPProcessNaNs3()
// =====

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPPProcessNaNs3(type1, type2, type3, op1, op2, op3, fpcr, fpexc);

// FPPProcessNaNs3()
// =====
// The boolean part of the return value says whether a NaN has been found and
// processed. The bits(N) part is only relevant if it has and supplies the
// result of the operation.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(boolean, bits(N)) FPPProcessNaNs3(FPType type1, FPType type2, FPType type3,
                                   bits(N) op1, bits(N) op2, bits(N) op3,
                                   FPCRType fpcr, boolean fpexc)

    assert N IN {16,32,64};
    bits(N) result;
    boolean op1_nan = type1 IN {FPType_SNaN, FPType_QNaN};
    boolean op2_nan = type2 IN {FPType_SNaN, FPType_QNaN};
    boolean op3_nan = type3 IN {FPType_SNaN, FPType_QNaN};

    boolean altfp = HaveAltFP() && !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 = FPPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op2_nan && (op1_nan || op3_nan) then
        // <n> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op2, fpcr, fpexc);
    elsif altfp && op3_nan && op1_nan then
        // <m> register NaN selected
        done = TRUE; result = FPPProcessNaN(type_nan, op3, fpcr, fpexc);
    elsif type1 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_SNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    elsif type1 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type1, op1, fpcr, fpexc);
    elsif type2 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type2, op2, fpcr, fpexc);
    elsif type3 == FPType_QNaN then
        done = TRUE; result = FPPProcessNaN(type3, op3, fpcr, fpexc);
    else
        done = FALSE; result = Zeros(); // 'Don't care' result

    return (done, result);
```



```

// FPrecipEstimate()
// =====

bits(N) FPrecipEstimate(bits(N) operand, FPCRTYPE fpcr_in)

assert N IN {16,32,64};
FPCRTYPE fpcr = fpcr_in;
bits(N) result;
boolean overflow_to_inf;
// When using alternative floating-point behaviour, do not generate
// floating-point exceptions, flush denormal input and output to zero,
// and use RNE rounding mode.
boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
boolean fpxc = !altfp;
if altfp then fpcr.<FIZ,FZ> = '11';
if altfp then fpcr.RMode = '00';

(fptype,sign,value) = FPUnpack(operand, fpcr, fpxc);

FPRounding rounding = FPRoundingMode(fpcr);
if fptype == FPTYPE\_SNaN || fptype == FPTYPE\_0NaN then
    result = FPProcessNaN(fptype, operand, fpcr, fpxc);
elseif fptype == FPTYPE\_Infinity then
    result = FPZero(sign);
elseif fptype == FPTYPE\_Zero then
    result = FPInfinity(sign);
    if fpxc 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) else FPMaxNormal(sign);
    if fpxc 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);

    // Flush-to-zero never generates a trapped exception.
    if UsingAArch32() then
        FPSCR.UFC = '1';
    else
        if fpxc then FPSR.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;
boolean increasedprecision = N==32 && HaveFeatRPRES() && 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
        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
        real real_val = Real(2^25)/Real(a);
        r = RoundDown(real_val);
        real error = real_val - Real(r);
        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, FPCRTType fpcr_in)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;
    integer esize;
    case N of
        when 16 esize = 5;
        when 32 esize = 8;
        when 64 esize = 11;

    bits(N)          result;
    bits(esize)      exp;
    bits(esize)      max_exp;
    bits(N-(esize+1)) frac = Zeros();

    boolean altfp = HaveAltFP() && fpcr.AH == '1';
    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);

    case N of
        when 16 exp = op<10+esize-1:10>;
        when 32 exp = op<23+esize-1:23>;
        when 64 exp = op<52+esize-1:52>;

    max_exp = Ones(esize) - 1;

    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr, fpexc);
    else
        if IsZero(exp) then           // Zero and denormals
            result = sign:max_exp:frac;
        else                          // Infinities and normals
            result = sign:NOT(exp):frac;

    return result;
```

Library pseudocode for shared/functions/float/fpround/FPRound

```
// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.

bits(N) FPRound(real op, FPCRTType fpcr_in, FPRounding rounding)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean fpxc = TRUE;    // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpxc);

// FPRound()
// =====
// Used by data processing and int/fixed <-> FP conversion instructions.
// For half-precision data it ignores AHP, and observes FZ16.
//
// The 'fpcr' argument supplies FPCR control bits and 'fpxc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

bits(N) FPRound(real op, FPCRTType fpcr_in, FPRounding rounding, boolean fpxc)
    FPCRTType fpcr = fpcr_in;
    fpcr.AHP = '0';
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpxc);

// FPRound()
// =====

bits(N) FPRound(real op, FPCRTType fpcr)
    return FPRound(op, fpcr, FPRoundingMode(fpcr));
```



```

// FPRoundBase()
// =====

bits(N) FPRoundBase(real op, FPCRTYPE fpcr, FPRounding rounding, boolean isbfloat16)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

// 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.

bits(N) FPRoundBase(real op, FPCRTYPE fpcr, FPRounding rounding,
    boolean isbfloat16, boolean fpexc)

    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.
    integer minimum_exp;
    integer F;
    integer E;
    if N == 16 then
        minimum_exp = -14; E = 5; F = 10;
    elsif N == 32 && isbfloat16 then
        minimum_exp = -126; E = 8; F = 7;
    elsif N == 32 then
        minimum_exp = -126; E = 8; F = 23;
    else // N == 64
        minimum_exp = -1022; E = 11; F = 52;

    // Split value into sign, unrounded mantissa and exponent.
    bit sign;
    real mantissa;
    if op < 0.0 then
        sign = '1'; mantissa = -op;
    else
        sign = '0'; mantissa = op;
    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;

    // 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 = HaveAltFP() && 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);

    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'.
if !altfp && biased_exp == 0 && (error != 0.0 || fpcr.UFE == '1') 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_TIEEVEN
            round_up_unconstrained = (error_unconstrained > 0.5 ||
                (error_unconstrained == 0.5 && int_mant_unconstrained<0> == '1'));
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        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);
        elsif error != 0.0 || fpcr.UFE == '1' then
            if fpexc then FPProcessException(FPExc_Underflow, fpcr);
else // altfp == FALSE
    case rounding of
        when FPRounding_TIEEVEN
            round_up = (error > 0.5 || (error == 0.5 && int_mant<0> == '1'));
            overflow_to_inf = TRUE;
        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 then
    int_mant<0> = '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) else FPMMaxNormal(sign);
        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 <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

bits(N) FPRoundCV(real op, FPCRTType fpcr_in, FPRounding rounding)
    FPCRTType fpcr = fpcr_in;
    fpcr.FZ16 = '0';
    boolean fpexc = TRUE; // Generate floating-point exceptions
    boolean isbfloat16 = FALSE;
    return FPRoundBase(op, fpcr, rounding, isbfloat16, fpexc);

```

Library pseudocode for shared/functions/float/fprounding/FPRounding

```

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(FPCRTType 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, FPCRTYPE 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 = HaveAltFP\(\) && !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 == FPTYPE\_SNaN || fptype == FPTYPE\_0NaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTYPE\_Infinity then
        result = FPInfinity(sign);
    elsif fptype == FPTYPE\_Zero then
        result = FPZero(sign);
    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);
        else
            result = FPRound(real_result, fpcr, FPRounding\_ZERO);

        // Generate inexact exceptions.
        if error != 0.0 && exact then
            FPProcessException(FPExc\_Inexact, fpcr);

    return result;
```



```

// FPRoundIntN()
// =====

bits(N) FPRoundIntN(bits(N) op, FPCRTType 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 = HaveAltFP() && !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, FPTType\_0NaN, 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);
  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);

```

```
    else
        result = FPRound(real_result, fpcr, FPRounding\_ZERO);

// Generate inexact exceptions.
if error != 0.0 then
    FPProcessException(FPExc\_Inexact, fpcr);

return result;
```



```

// FPRsqrtEstimate()
// =====

bits(N) FPRsqrtEstimate(bits(N) operand, FPCRTType fpcr_in)

    assert N IN {16,32,64};
    FPCRTType fpcr = fpcr_in;

    // When using alternative floating-point behaviour, do not generate
    // floating-point exceptions and flush denormal input to zero.
    boolean altfp = HaveAltFP() && !UsingAArch32() && fpcr.AH == '1';
    boolean fpxc = !altfp;
    if altfp then fpcr.<FIZ,FZ> = '11';

    (fptype,sign,value) = FPUnpack(operand, fpcr, fpxc);

    bits(N) result;
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, operand, fpcr, fpxc);
    elsif fptype == FPTType\_Zero then
        result = FPInfinity(sign);
        if fpxc then FPProcessException(FPExc\_DivideByZero, fpcr);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr);
        if fpxc then FPProcessException(FPExc\_InvalidOp, fpcr);
    elsif fptype == FPTType\_Infinity then
        result = FPZero('0');
    else
        // Scale to a fixed-point value in the range 0.25 <= x < 1.0 in steps of 512, with the
        // evenness or oddness of the exponent unchanged, and calculate result exponent.
        // Scaled value has copied sign bit, exponent = 1022 or 1021 = double-precision
        // biased version of -1 or -2, fraction = original fraction extended with zeros.

        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;
        boolean increasedprecision = N==32 && HaveFeatRPRES() && 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/fprsqrtestimate/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);

        real_val = Sqrt(real_val); // This number will lie in the range of 32 to 64
        // Round to nearest even for a DP float number
        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, FPCRTType fpcr)

    assert N IN {16,32,64};
    (fptype,sign,value) = FPUntpack(op, fpcr);

    bits(N) result;
    if fptype == FPTType\_SNaN || fptype == FPTType\_QNaN then
        result = FPProcessNaN(fptype, op, fpcr);
    elsif fptype == FPTType\_Zero then
        result = FPZero(sign);
    elsif fptype == FPTType\_Infinity && sign == '0' then
        result = FPInfinity(sign);
    elsif sign == '1' then
        result = FPDefaultNaN(fpcr);
        FPProcessException(FPExc\_InvalidOp, fpcr);
    else
        result = FPRound(Sqrt(value), 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, FPCRTType fpcr)

    assert N IN {16,32,64};
    rounding = FPRoundingMode(fpcr);

    (type1,sign1,value1) = FPUntpack(op1, fpcr);
    (type2,sign2,value2) = FPUntpack(op2, fpcr);

    (done,result) = FPProcessNaNs(type1, type2, op1, op2, fpcr);
    if !done then
        inf1 = (type1 == FPTType\_Infinity);
        inf2 = (type2 == FPTType\_Infinity);
        zero1 = (type1 == FPTType\_Zero);
        zero2 = (type2 == FPTType\_Zero);

        if inf1 && inf2 && sign1 == sign2 then
            result = FPDefaultNaN(fpcr);
            FPProcessException(FPExc\_InvalidOp, fpcr);
        elsif (inf1 && sign1 == '0') || (inf2 && sign2 == '1') then
            result = FPInfinity('0');
        elsif (inf1 && sign1 == '1') || (inf2 && sign2 == '0') then
            result = FPInfinity('1');
        elsif zero1 && zero2 && sign1 == NOT(sign2) then
            result = FPZero(sign1);
        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);
            else
                result = FPRound(result_value, fpcr, rounding);

            FPProcessDenorms(type1, type2, N, fpcr);

    return result;
```

Library pseudocode for shared/functions/float/fpthree/FPThree

```
// FPThree()
// =====

bits(N) FPThree(bit sign)

    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, FPCRTYPE fpcr, FPRounding rounding)

    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 = HaveAltFP() && !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, FPCRType fpcr, boolean Is64)

    assert M == 64 && N == 32;

    // If FALSE, never generate Input Denormal floating-point exceptions.
    fpxc_idenorm = !(HaveAltFP()) && !UsingAArch32() && fpcr.AH == '1');

    // Unpack using fpcr to determine if subnormals are flushed-to-zero.
    (fptype,sign,value) = FPUunpack(op, fpcr, fpxc_idenorm);

    Z = '1';
    // If NaN, set cumulative flag or take exception.
    if fptype == FPTType\_SNaN || fptype == FPTType\_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 == FPTType\_Infinity then result = 0;

    return (result<N-1:0>, Z);
```

Library pseudocode for shared/functions/float/fptwo/FPTwo

```
// FPTwo()
// =====

bits(N) FPTwo(bit sign)

    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

```
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, FPCRTType fpcr_in)  
    FPCRTType fpcr = fpcr_in;  
    fpcr.AHP = '0';  
    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 and int/fixed <-> FP conversion instructions.  
// For half-precision data it ignores AHP, and observes FZ16.  
  
(FPType, bit, real) FPUnpack(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc)  
    FPCRTType fpcr = fpcr_in;  
    fpcr.AHP = '0';  
    (fp_type, sign, value) = FPUnpackBase(fpval, fpcr, fpexc);  
    return (fp_type, sign, value);
```



```

// FPUunpackBase()
// =====

(FPType, bit, real) FPUunpackBase(bits(N) fpval, FPCRTType fpcr)
    boolean fpexc = TRUE; // Generate floating-point exceptions
    (fp_type, sign, value) = FPUunpackBase(fpval, fpcr, fpexc);
    return (fp_type, sign, value);

// FPUunpackBase()
// =====
//
// 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 and 'fpexc' controls the
// generation of floating-point exceptions. Status information is updated
// directly in the FPSR where appropriate.

(FPType, bit, real) FPUunpackBase(bits(N) fpval, FPCRTType fpcr_in, boolean fpexc)

    assert N IN {16,32,64};

    FPCRTType fpcr = fpcr_in;

    boolean altfp = HaveAltFP() && !UsingAArch32();
    boolean fiz   = altfp && fpcr.FIZ == '1';
    boolean fz    = fpcr.FZ == '1' && !(altfp && fpcr.AH == '1');
    real value;
    bit sign;
    FPType fptype;

    if N == 16 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 then
        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 FPProcessException(FPExc\_InputDenorm, fpcr);
        else

```

```

        fptype = FPTType\_Denormal; value = 2.0^-126 * (Real(UInt(frac32)) * 2.0^-23);
elseif IsOnes(exp32) then
    if IsZero(frac32) then
        fptype = FPTType\_Infinity; value = 2.0^1000000;
    else
        fptype = if frac32<22> == '1' then FPTType\_QNaN else FPTType\_SNaN;
        value = 0.0;
    else
        fptype = FPTType\_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 = FPTType\_Zero; value = 0.0;
        elseif fz || fiz then // Flush-to-zero if FIZ==1 or AH,FZ==01
            fptype = FPTType\_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 = FPTType\_Denormal; value = 2.0^-1022 * (Real(UInt(frac64)) * 2.0^-52);
        elseif IsOnes(exp64) then
            if IsZero(frac64) then
                fptype = FPTType\_Infinity; value = 2.0^1000000;
            else
                fptype = if frac64<51> == '1' then FPTType\_QNaN else FPTType\_SNaN;
                value = 0.0;
        else
            fptype = FPTType\_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 <-> FP conversion instructions.
// For half-precision data ignores FZ16 and observes AHP.

(FPTType, bit, real) FPUnpackCV(bits(N) fpval, FPCRTType fpcr_in)
FPCRTType fpcr = fpcr_in;
fpcr.FZ16 = '0';
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)

    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/VFPExpandImm

```
// VFPExpandImm()
// =====

bits(N) VFPExpandImm(bits(8) imm8)

    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)
    integer unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);
    integer signed_sum = SInt(x) + SInt(y) + UInt(carry_in);
    bits(N) result = unsigned_sum<N-1:0>; // same value as signed_sum<N-1:0>
    bit n = result<N-1>;
    bit z = if IsZero(result) then '1' else '0';
    bit c = if UInt(result) == unsigned_sum then '0' else '1';
    bit v = if SInt(result) == signed_sum then '0' else '1';
    return (result, n:z:c:v);
```

Library pseudocode for shared/functions/interrupts/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,
};
```

Library pseudocode for shared/functions/interrupts/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 for storing to the program counter.  
  
bits(64) AArch64.BranchAddr(bits(64) vaddress)  
    assert !UsingAArch32();  
    msbit = AddrTop(vaddress, TRUE, PSTATE.EL);  
    if msbit == 63 then  
        return vaddress;  
    elsif (PSTATE.EL IN {EL0, EL1} || IsInHost()) && vaddress<msbit> == '1' then  
        return SignExtend(vaddress<msbit:0>);  
    else  
        return ZeroExtend(vaddress<msbit:0>);
```

Library pseudocode for shared/functions/memory/Acctype

```
enumeration AccType {AccType\_NORMAL, // Normal loads and stores  
    AccType\_STREAM, // Streaming loads and stores  
    AccType\_VEC, // Vector loads and stores  
    AccType\_VECSTREAM, // Streaming vector loads and stores  
    AccType\_UNPRIVSTREAM, // Streaming unprivileged loads and stores  
    AccType\_A32LSMD, // Load and store multiple  
    AccType\_ATOMIC, // Atomic loads and stores  
    AccType\_ATOMICRW,  
    AccType\_ORDERED, // Load-Acquire and Store-Release  
    AccType\_ORDEREDRW,  
    AccType\_ORDEREDATOMIC, // Load-Acquire and Store-Release with atomic access  
    AccType\_ORDEREDATOMICRW,  
    AccType\_ATOMICS64, // Atomic 64-byte loads and stores  
    AccType\_LIMITEDORDERED, // Load-LOAcquire and Store-LORelease  
    AccType\_UNPRIV, // Load and store unprivileged  
    AccType\_IFETCH, // Instruction fetch  
    AccType\_TTW, // Translation table walk  
    AccType\_NV2REGISTER, // MRS/MSR instruction used at EL1 and which is  
    // converted to a memory access that uses the  
    // EL2 translation regime  
  
    // Other operations  
    AccType\_DC, // Data cache maintenance  
    AccType\_IC, // Instruction cache maintenance  
    AccType\_DCZVA, // DC ZVA instructions  
    AccType\_ATPAN, // Address translation with PAN permission checks  
    AccType\_AT}; // Address translation
```

Library pseudocode for shared/functions/memory/AccessDescriptor

```
type AccessDescriptor is (  
    MPAMInfo mpam,  
    AccType acctype)
```

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.

integer 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/Allocation

```
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(AccType acctype)
    boolean bigend;
    if HaveNV2Ext() && acctype == AccType_NV2REGISTER then
        return SCTLRL_EL2.EE == '1';

    if UsingAArch32() then
        bigend = (PSTATE.E != '0');
    elsif PSTATE.EL == EL0 then
        bigend = (SCTLR[.E0E != '0');
    else
        bigend = (SCTLR[.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};
    integer half = width DIV 2;
    if width == 8 then return value;
    return BigEndianReverse(value<half-1:0>) : BigEndianReverse(value<width-1:half>);
```

Library pseudocode for shared/functions/memory/Cacheability

```
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/CreateAccessDescriptor

```
// CreateAccessDescriptor()
// =====

AccessDescriptor CreateAccessDescriptor(AccType acctype)
    AccessDescriptor accdesc;
    accdesc.acctype = acctype;
    accdesc.mpam = GenMPAMcurEL(acctype);
    return accdesc;
```

Library pseudocode for shared/functions/memory/DataMemoryBarrier

```
DataMemoryBarrier(MBReqDomain domain, MBReqTypes types);
```

Library pseudocode for shared/functions/memory/DataSynchronizationBarrier

```
DataSynchronizationBarrier(MBReqDomain domain, MBReqTypes types, boolean nXS);
```

Library pseudocode for shared/functions/memory/DeviceType

```
enumeration DeviceType {DeviceType_GRE, DeviceType_nGRE, DeviceType_nGnRE, DeviceType_nGnRnE};
```

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 HavePACExt() then
                tbid = if address<55> == '1' then TCR_EL1.TBID1 else TCR_EL1.TBID0;
        when EL2
            if HaveVirtHostExt() && ELIsInHost(el) then
                tbi = if address<55> == '1' then TCR_EL2.TBI1 else TCR_EL2.TBI0;
                if HavePACExt() then
                    tbid = if address<55> == '1' then TCR_EL2.TBID1 else TCR_EL2.TBID0;
            else
                tbi = TCR_EL2.TBI;
                if HavePACExt() then tbid = TCR_EL2.TBID;
        when EL3
            tbi = TCR_EL3.TBI;
            if HavePACExt() then tbid = TCR_EL3.TBID;

    return (if tbi == '1' && (!HavePACExt() || 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 HaveVirtHostExt() && 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/Fault

```
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_AsyncParity,
    Fault_AsyncExternal,
    Fault_Debug,
    Fault_TLBConflict,
    Fault_BranchTarget,
    Fault_HWUpdateAccessFlag,
    Fault_Lockdown,
    Fault_Exclusive,
    Fault_ICacheMaint};
```

Library pseudocode for shared/functions/memory/FaultRecord

```
type FaultRecord is (Fault    statuscode, // Fault Status
    AccType acctype, // Type of access that faulted
    FullAddress ipaddress, // Intermediate physical address
    boolean s2fslwalk, // Is on a Stage 1 translation table walk
    boolean write, // TRUE for a write, FALSE for a read
    integer level, // For translation, access flag and Permission faults
    bit extflag, // IMPLEMENTATION DEFINED syndrome for External aborts
    boolean secondstage, // Is a Stage 2 abort
    bits(4) domain, // Domain number, AArch32 only
    bits(2) errortype, // [Armv8.2 RAS] AArch32 AET or AArch64 SET
    bits(4) debugmoe) // Debug method of entry, from AArch32 only
```

Library pseudocode for shared/functions/memory/FullAddress

```
type FullAddress is (  
    PASpace paspace,  
    bits(52) address  
)
```

Library pseudocode for shared/functions/memory/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 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/IsDataAccess

```
// IsDataAccess()  
// =====  
// Return TRUE if access is to data memory.  
  
boolean IsDataAccess(AccType acctype)  
    return !(acctype IN {AccType_IFETCH, AccType_TTW,  
                        AccType_DC, AccType_IC,  
                        AccType_AT, AccType_ATPAN  
                        });
```

Library pseudocode for shared/functions/memory/MBReqDomain

```
enumeration MBReqDomain    {MBReqDomain_Nonshareable, MBReqDomain_InnerShareable,  
                            MBReqDomain_OuterShareable, MBReqDomain_FullSystem};
```

Library pseudocode for shared/functions/memory/MBReqTypes

```
enumeration MBReqTypes    {MBReqTypes_Reads, MBReqTypes_Writes, MBReqTypes_All};
```

Library pseudocode for shared/functions/memory/MPAM

```
type PARTIDtype = bits(16);  
type PMGtype = bits(8);  
  
enumeration PARTIDspaceType {  
    PIdSpace_Secure,  
    PIdSpace_NonSecure  
};  
  
type MPAMinfo is (  
    PARTIDspaceType mpam_sp,  
    PARTIDtype partid,  
    PMGtype pmg  
)
```

Library pseudocode for shared/functions/memory/MemAttrHints

```
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/MemType

```
enumeration MemType {MemType_Normal, MemType_Device};
```

Library pseudocode for shared/functions/memory/MemoryAttributes

```
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  
    boolean tagged, // Tagged access  
    bit xs // XS attribute  
)
```

Library pseudocode for shared/functions/memory/PASpace

```
enumeration PASpace {  
    PAS_NonSecure,  
    PAS_Secure,  
};
```

Library pseudocode for shared/functions/memory/Permissions

```
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 pxn, // Stage 1 privileged execute-never  
    bits(2) s2ap, // Stage 2 access permissions  
    bit s2uxnx, // Stage 2 extended execute-never  
    bit s2xn // Stage 2 execute-never  
)
```

Library pseudocode for shared/functions/memory/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

```
type PhysMemRetStatus is (Fault      statuscode,      // Fault Status
                          bit        extflag,          // IMPLEMENTATION DEFINED
                          bits(2)    errortype,        // syndrome for External aborts
                                          // optional error state
                                          // returned on a physical
                                          // memory access
                          bits(64)    store64bstatus, // status of 64B store
                          AccType    acctype)           // Type of access that faulted
```

Library pseudocode for shared/functions/memory/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

```
enumeration PrefetchHint {Prefetch_READ, Prefetch_WRITE, Prefetch_EXEC};
```

Library pseudocode for shared/functions/memory/Shareability

```
enumeration Shareability {
    Shareability_NSH,
    Shareability_ISH,
    Shareability_OSH
};
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToPA

```
SpeculativeStoreBypassBarrierToPA();
```

Library pseudocode for shared/functions/memory/SpeculativeStoreBypassBarrierToVA

```
SpeculativeStoreBypassBarrierToVA();
```

Library pseudocode for shared/functions/memory/Tag

```
constant integer LOG2_TAG_GRANULE = 4;
constant integer TAG_GRANULE = 1 << LOG2\_TAG\_GRANULE;
```

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 = DefaultPARTID;
    DefaultInfo.pmg     = DefaultPMG;
    return DefaultInfo;
```

Library pseudocode for shared/functions/mpam/DefaultPARTID

```
constant PARTIDtype DefaultPARTID = 0<15:0>;
```

Library pseudocode for shared/functions/mpam/DefaultPMG

```
constant PMGtype DefaultPMG = 0<7:0>;
```

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(AccType acctype)
    bits(2) mpamEL;
    boolean validEL = FALSE;
    SecurityState security = CurrentSecurityState();
    boolean InD = 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 == AccType_NV2REGISTER then EL2 else PSTATE.EL;
        validEL = TRUE;
    case acctype of
        when AccType_IFETCH, AccType_IC
            InD = TRUE;
        otherwise
            // Other access types are DATA accesses
            InD = FALSE;
    if !validEL then
        return DefaultMPAMinfo(pspace);
    if HaveEMPAMExt() && security == SS_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            pspace = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFULT == '1' then
            return DefaultMPAMinfo(pspace);
    if !MPAMisEnabled() then
        return DefaultMPAMinfo(pspace);
    else
        return genMPAM(mpamEL, InD, pspace);
```

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);
    integer vpmrmax = UInt(MPAMIDR_EL1.VPMR_MAX);

    // vpartid_max is largest vpartid supported
    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?
    elsif 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 = DefaultPARTID;
        err = TRUE;

    // Check that the physical PARTID is in-range.
    // This physical PARTID came from a virtual mapping entry.
    integer partid_max = UInt(MPAMIDR_EL1.PARTID_MAX);
    if UInt(ret) > partid_max then
        // Out of range, so return default physical PARTID
        ret = DefaultPARTID;
        err = TRUE;
    return (ret, err);
```

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' &&
              (HCR_EL2.E2H == '0' || HCR_EL2.TGE == '0')) ||
             (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\_Secure
            return PIdSpace\_Secure;
        otherwise
            Unreachable\(\);
```

Library pseudocode for shared/functions/mpam/genMPAM

```
// genMPAM()
// =====
// Returns MPAMinfo for exception level el.
// If InD 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.
// Produces a PARTID in PARTID space pspace.

MPAMinfo genMPAM(bits(2) el, boolean InD, 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.
    boolean gstplk = (el == EL0 && EL2Enabled\(\) &&
                     MPAMHCR_EL2.GSTAPP_PLK == '1' &&
                     HCR_EL2.TGE == '0');
    bits(2) eff_el = if gstplk then EL1 else el;
    (partidel, perr) = genPARTID(eff_el, InD);
    PMGtype groupel = genPMG(eff_el, InD, perr);
    returninfo.mpam_sp = pspace;
    returninfo.partid = partidel;
    returninfo.pmg = groupel;
    return returninfo;
```

Library pseudocode for shared/functions/mpam/genMPAMel

```
// genMPAMel()
// =====
// Returns MPAMinfo for specified EL in the current security state.
// InD is TRUE for instruction access and FALSE otherwise.

MPAMinfo genMPAMel(bits(2) el, boolean InD)
    SecurityState security = SecurityStateAtEL(el);
    PARTIDspaceType space = PARTIDspaceFromSS(security);
    boolean use_default = !(HaveMPAMExt() && MPAMisEnabled());
    if HaveEMPAMExt() && security == SS_Secure then
        if MPAM3_EL3.FORCE_NS == '1' then
            space = PIdSpace_NonSecure;
        if MPAM3_EL3.SDEFALT == '1' then
            use_default = TRUE;
    if !use_default then
        return genMPAM(el, InD, space);
    else
        return DefaultMPAMinfo(space);
```

Library pseudocode for shared/functions/mpam/genPARTID

```
// genPARTID()
// =====
// Returns physical PARTID and error boolean for exception level el.
// If InD is TRUE then PARTID is from MPAMel_ELx.PARTID_I and
// otherwise from MPAMel_ELx.PARTID_D.

(PARTIDtype, boolean) genPARTID(bits(2) el, boolean InD)
    PARTIDtype partidel = getMPAM_PARTID(el, InD);
    PARTIDtype partid_max = MPAMIDR_EL1.PARTID_MAX;
    if UInt(partidel) > UInt(partid_max) then
        return (DefaultPARTID, 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 (InD).
// If PARTID generation (genPARTID) encountered an error, genPMG() should be
// called with partid_err as TRUE.

PMGtype genPMG(bits(2) el, boolean InD, boolean partid_err)
    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 DefaultPMG;
    PMGtype groupel = getMPAM_PMG(el, InD);
    if UInt(groupel) <= pmg_max then
        return groupel;
    return DefaultPMG;
```

Library pseudocode for shared/functions/mpam/getMPAM_PARTID

```
// getMPAM_PARTID()
// =====
// Returns a PARTID from one of the MPAMn_ELx registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PARTID_I field of that
// register. Otherwise, selects the PARTID_D field.

PARTIDtype getMPAM_PARTID(bits(2) MPAMn, boolean InD)
    PARTIDtype partid;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when '11' partid = MPAM3_EL3.PARTID_I;
            when '10' partid = if el2avail then MPAM2_EL2.PARTID_I else Zeros();
            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 el2avail then MPAM2_EL2.PARTID_D else Zeros();
                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 registers.
// MPAMn selects the MPAMn_ELx register used.
// If InD is TRUE, selects the PMG_I field of that
// register. Otherwise, selects the PMG_D field.

PMGtype getMPAM_PMG(bits(2) MPAMn, boolean InD)
    PMGtype pmg;
    boolean el2avail = EL2Enabled();

    if InD then
        case MPAMn of
            when '11' pmg = MPAM3_EL3.PMG_I;
            when '10' pmg = if el2avail then MPAM2_EL2.PMG_I else Zeros();
            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 el2avail then MPAM2_EL2.PMG_D else Zeros();
                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/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;
    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
    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 EL2Enabled() && !ELUsingAArch32(EL2) && HCR_EL2.<E2H, TGE> == '11' then
        if TCR_EL2.A1 == '1' then
            return TTBR1_EL2.ASID;
        else
            return TTBR0_EL2.ASID;

    elsif !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

```
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

```
enumeration RestrictType {
    RestrictType_DataValue,
    RestrictType_ControlFlow,
    RestrictType_CachePrefetch
};
```

Library pseudocode for shared/functions/predictionrestrict/TargetSecurityState

```
// TargetSecurityState()
// =====
// Decode the target security state for the prediction context.

SecurityState TargetSecurityState(bit NS)
    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;
```

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);
    else
        assert N == 64 && !UsingAArch32();
        bits(64) target_vaddress = AArch64.BranchAddr(target<63:0>);

        _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);
    else
        assert N == 64 && !UsingAArch32();
        _PC = target<63:0>;
    return;
```

Library pseudocode for shared/functions/registers/BranchType

```
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_RESET,     // Reset
    BranchType_UNKNOWN};  // Other
```

Library pseudocode for shared/functions/registers/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

```
// Return address of the sequentially next instruction.
bits(N) NextInstrAddr();
```

Library pseudocode for shared/functions/registers/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()
    assert N == 64 || (N == 32 && UsingAArch32());
    return _PC<N-1:0>;
```

Library pseudocode for shared/functions/registers/_PC

```
bits(64) _PC;
```

Library pseudocode for shared/functions/registers/_R

```
array bits(64) _R[0..30];
```

Library pseudocode for shared/functions/registers/_V

```
array bits(128) _V[0..31];
```

Library pseudocode for shared/functions/sysregisters/SPSR

```
// SPSR[] - non-assignment form
// =====

bits(N) SPSR[]
  bits(N) result;
  if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
      when M32_FIQ      result = SPSR_fiq<N-1:0>;
      when M32_IRQ      result = SPSR_irq<N-1:0>;
      when M32_Svc      result = SPSR_svc<N-1:0>;
      when M32_Monitor  result = SPSR_mon<N-1:0>;
      when M32_Abort    result = SPSR_abt<N-1:0>;
      when M32_Hyp      result = SPSR_hyp<N-1:0>;
      when M32_Undef    result = SPSR_und<N-1:0>;
      otherwise        Unreachable();
  else
    assert N == 64;
    case PSTATE.EL of
      when EL1          result = SPSR_EL1<N-1:0>;
      when EL2          result = SPSR_EL2<N-1:0>;
      when EL3          result = SPSR_EL3<N-1:0>;
      otherwise        Unreachable();
  return result;

// SPSR[] - assignment form
// =====

SPSR[] = bits(N) value
  if UsingAArch32() then
    assert N == 32;
    case PSTATE.M of
      when M32_FIQ      SPSR_fiq = ZeroExtend(value);
      when M32_IRQ      SPSR_irq = ZeroExtend(value);
      when M32_Svc      SPSR_svc = ZeroExtend(value);
      when M32_Monitor  SPSR_mon = ZeroExtend(value);
      when M32_Abort    SPSR_abt = ZeroExtend(value);
      when M32_Hyp      SPSR_hyp = ZeroExtend(value);
      when M32_Undef    SPSR_und = ZeroExtend(value);
      otherwise        Unreachable();
  else
    assert N == 64;
    case PSTATE.EL of
      when EL1          SPSR_EL1 = ZeroExtend(value);
      when EL2          SPSR_EL2 = ZeroExtend(value);
      when EL3          SPSR_EL3 = ZeroExtend(value);
      otherwise        Unreachable();
  return;
```

Library pseudocode for shared/functions/system/ArchVersion

```
enumeration ArchVersion {
  ARMv8p0
  , ARMv8p1
  , ARMv8p2
  , ARMv8p3
  , ARMv8p4
  , ARMv8p5
  , ARMv8p6
  , ARMv8p7
  , ARMv8p8
};
```

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 HaveBTIExt() && !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
        bits(64) pc = ThisInstrAddr();
        AArch64.BranchTargetException(pc<51:0>);

    boolean branch_instr = AArch64.ExecutingBR0rBLR0rRetInstr();
    boolean bti_instr    = AArch64.ExecutingBTIInstr();

    // PSTATE.BTYPE defaults to 00 for instructions that do not explicitly set BTYPE.
    if !(branch_instr || bti_instr) then
        BTypeNext = '00';
```

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/ClearPendingPhysicalSError

```
// Clear a pending physical SError interrupt.
ClearPendingPhysicalSError();
```

Library pseudocode for shared/functions/system/ClearPendingVirtualSError

```
// Clear a pending virtual SError interrupt.
ClearPendingVirtualSError();
```

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();
```

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

```
enumeration DSBAlias {DSBAlias_SSBB, DSBAlias_PSSBB, DSBAlias_DSB};
```

Library pseudocode for shared/functions/system/EL0

```
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 Secure state when Secure EL2 is implemented and enabled, or
// - when EL3 is not implemented.

boolean EL2Enabled()
    return HaveEL(EL2) && (!HaveEL(EL3) || SCR_GEN[].NS == '1' || IsSecureEL2Enabled());
```

Library pseudocode for shared/functions/system/EL3SDDTrapPriority

```
// EL3SDDTrapPriority()
// =====
// Returns TRUE if in Debug state, EDSCR.SDD is set, and an EL3 trap by an
// EL3 control register has the priority of the original trap exception.

boolean EL3SDDTrapPriority()
    return (Halted() && EDSCR.SDD == '1' &&
        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
    case mode of
        when M32_Monitor
            el = EL3;
        when M32_Hyp
            el = EL2;
            valid = valid && (!HaveEL(EL3) || SCR_GEN[].NS == '1');
        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 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>;
        if !HaveAArch64() then // No AArch64 support
            valid = FALSE;
        elsif !HaveEL(el) then // Exception level not implemented
            valid = FALSE;
        elsif spsr<1> == '1' then // M[1] must be 0
            valid = FALSE;
        elsif el == EL0 && spsr<0> == '1' then // for EL0, M[0] must be 0
            valid = FALSE;
        elsif 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;
    elsif 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 !HaveVirtHostExt() || ELUsingAArch32(EL2) then
        return FALSE;
    case el of
        when EL3
            return FALSE;
        when EL2
            return EL2Enabled() && HCR_EL2.E2H == '1';
        when EL1
            return FALSE;
        when EL0
            return EL2Enabled() && HCR_EL2.<E2H, 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()
// =====

(boolean,boolean) ELStateUsingAArch32K(bits(2) el, boolean secure)
// 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.
if !HaveAArch32EL(el) then
    return (TRUE, FALSE); // Exception level is using AArch64
elseif secure && el == EL2 then
    return (TRUE, FALSE); // Secure EL2 is using AArch64
elseif !HaveAArch64() then
    return (TRUE, TRUE); // Highest Exception level, and therefore all levels are using AArch64

// Remainder of function deals with the interprocessing cases when highest Exception level is using AArch64

boolean aarch32 = boolean UNKNOWN;
boolean known = TRUE;

aarch32_below_el3 = HaveEL(EL3) && SCR_EL3.RW == '0' && (!secure || !HaveSecureEL2Ext() || SCR_EL3.EEL2 == '1');
aarch32_at_el1 = (aarch32_below_el3 || (HaveEL(EL2) && ((HaveSecureEL2Ext() && SCR_EL3.EEL2 == '1') || !secure) && HCR_EL2.E2H == '1' && HCR_EL2.TGE == '1' && HaveVirtHostExt())) || (el == EL0 && PSTATE.EL == EL0 && PSTATE.nRW == '1');

if el == EL0 && !aarch32_at_el1 then // Only know if EL0 using AArch32 from PSTATE
    if PSTATE.EL == EL0 then
        aarch32 = PSTATE.nRW == '1'; // EL0 controlled by PSTATE
    else
        known = FALSE; // EL0 state is UNKNOWN
else
    aarch32 = (aarch32_below_el3 && el != EL3) || (aarch32_at_el1 && el IN {EL1, EL0});

if !known then aarch32 = boolean UNKNOWN;
return (known, aarch32);
```

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/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

```
// Terminate processing of the current instruction.  
EndOfInstruction();
```

Library pseudocode for shared/functions/system/EnterLowPowerState

```
// PE enters a low-power state.  
EnterLowPowerState();
```

Library pseudocode for shared/functions/system/EventRegister

```
bits(1) EventRegister;
```

Library pseudocode for shared/functions/system/ExceptionalOccurrenceTargetState

```
enumeration ExceptionalOccurrenceTargetState {  
    AArch32_NonDebugState,  
    AArch64_NonDebugState,  
    DebugState  
};
```

Library pseudocode for shared/functions/system/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

```
// Returns FP exceptions accumulated by the PE.  
bits(8) GetAccumulatedFPExceptions();
```

Library pseudocode for shared/functions/system/GetPSRFromPSTATE

```
// GetPSRFromPSTATE()
// =====
// Return a PSR value which represents the current PSTATE

bits(N) GetPSRFromPSTATE(ExceptionalOccurrenceTargetState targetELState)
  if UsingAArch32() && (targetELState IN {AArch32_NonDebugState, DebugState}) then
    assert N == 32;
  else
    assert N == 64;
  bits(N) spsr = Zeros();
  spsr<31:28> = PSTATE.<N,Z,C,V>;
  if HavePANExt() then spsr<22> = PSTATE.PAN;
  spsr<20> = PSTATE.IL;
  if PSTATE.nRW == '1' then // AArch32 state
    spsr<27> = PSTATE.Q;
    spsr<26:25> = PSTATE.IT<1:0>;
    if HaveSSBSExt() then spsr<23> = PSTATE.SSBS;
    if HaveDITExt() 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 HaveMTEExt() then spsr<25> = PSTATE.TC0;
    if HaveDITExt() then spsr<24> = PSTATE.DIT;
    if HaveUAOExt() then spsr<23> = PSTATE.UAO;
    spsr<21> = PSTATE.SS;
    if HaveFeatNMI() then spsr<13> = PSTATE.ALLINT;
    if HaveSSBSExt() then spsr<12> = PSTATE.SSBS;
    if HaveBTIExt() 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/HasArchVersion

```
// HasArchVersion()
// =====
// Returns TRUE if the implemented architecture includes the extensions defined in the specified
// architecture version.

boolean HasArchVersion(ArchVersion version)
  return version == ARMv8p0 || boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch32

```
// HaveAArch32()
// =====
// Return TRUE if AArch32 state is supported at at least EL0.

boolean HaveAArch32()
  return boolean IMPLEMENTATION_DEFINED "AArch32 state is supported at at least EL0";
```

Library pseudocode for shared/functions/system/HaveAArch32EL

```
// HaveAArch32EL()
// =====

boolean HaveAArch32EL(bits(2) el)
// Return TRUE if Exception level 'el' supports AArch32 in this implementation
if !HaveEL(el) then
    return FALSE; // The Exception level is not implemented
elseif !HaveAArch32() then
    return FALSE; // No Exception level can use AArch32
elseif !HaveAArch64() then
    return TRUE; // All Exception levels are using AArch32
elseif el == HighestEL() then
    return FALSE; // The highest Exception level is using AArch64
elseif el == EL0 then
    return TRUE; // EL0 must support using AArch32 if any AArch32
return boolean IMPLEMENTATION_DEFINED;
```

Library pseudocode for shared/functions/system/HaveAArch64

```
// HaveAArch64()
// =====
// Return TRUE if the highest Exception level is using AArch64 state.

boolean HaveAArch64()
    return boolean IMPLEMENTATION_DEFINED "Highest EL using AArch64";
```

Library pseudocode for shared/functions/system/HaveEL

```
// HaveEL()
// =====
// Return TRUE if Exception level 'el' is supported

boolean HaveEL(bits(2) el)
    if el IN {EL1,EL0} then
        return TRUE; // EL1 and EL0 must exist
    return boolean IMPLEMENTATION_DEFINED;
```

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) && HaveSecureEL2Ext();
            else
                return HaveEL(EL2);
        otherwise
            return (HaveEL(EL3) ||
                (secure == boolean IMPLEMENTATION_DEFINED "Secure-only implementation"));
```

Library pseudocode for shared/functions/system/HaveFP16Ext

```
// HaveFP16Ext()
// =====
// Return TRUE if FP16 extension is supported

boolean HaveFP16Ext()
    return boolean IMPLEMENTATION_DEFINED;
```

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_DGH

```
// Provides a hint to close any gathering occurring within the micro-architecture.
Hint_DGH();
```

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 or, for WFET, a local
// timeout event is generated when the virtual timer value equals or
// exceeds the supplied threshold value.

Hint_WFE(integer localtimeout, WfxType wfxtype)
  if IsEventRegisterSet() then
    ClearEventRegister();
  elseif HaveFeatWfXT() && LocalTimeoutEvent(localtimeout) then
    // No further operation if the local timeout has expired.
    EndOfInstruction();
  else
    bits(2) target_el;
    trap = FALSE;
    if PSTATE.EL == EL0 then
      // Check for traps described by the OS which may be EL1 or EL2.
      if HaveTWEExt() then
        sctlr = SCTLR[];
        trap = sctlr.nTWE == '0';
        target_el = EL1;
      else
        AArch64.CheckForWfXTrap(EL1, wfxtype);
    if !trap && PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
      // Check for traps described by the Hypervisor.
      if HaveTWEExt() then
        trap = HCR_EL2.TWE == '1';
        target_el = EL2;
      else
        AArch64.CheckForWfXTrap(EL2, wfxtype);

    if !trap && HaveEL(EL3) && PSTATE.EL != EL3 then
      // Check for traps described by the Secure Monitor.
      if HaveTWEExt() then
        trap = SCR_EL3.TWE == '1';
        target_el = EL3;
      else
        AArch64.CheckForWfXTrap(EL3, wfxtype);

    if trap && PSTATE.EL != EL3 then
      (delay_enabled, delay) = WFETrapDelay(target_el); // (If trap delay is enabled, Delay amount)
      if !WaitForEventUntilDelay(delay_enabled, delay) then
        // Event did not arrive before delay expired
        AArch64.WfXTrap(wfxtype, target_el); // Trap WFE
  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 or, for WFIT, a local timeout
// event is generated when the virtual timer value equals or exceeds the
// supplied threshold value.

Hint_WFI(integer localtimeout, WfxType wfxtype)
  if InterruptPending() || (HaveFeatWfxt() && LocalTimeoutEvent(localtimeout)) then
    // No further operation if an interrupt is pending or the local timeout has expired.
    EndOfInstruction();
  else
    if PSTATE.EL == EL0 then
      // Check for traps described by the OS.
      AArch64.CheckForWfxTrap(EL1, wfxtype);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
      // Check for traps described by the Hypervisor.
      AArch64.CheckForWfxTrap(EL2, wfxtype);
    if HaveEL(EL3) && PSTATE.EL != EL3 then
      // Check for traps described by the Secure Monitor.
      AArch64.CheckForWfxTrap(EL3, wfxtype);
    WaitForInterrupt(localtimeout);
```

Library pseudocode for shared/functions/system/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

```
// 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 HaveEL(EL2) && target == EL1 && HCR_EL2.TGE == '1' then
        if (!IsSecureBelowEL3() || IsSecureEL2Enabled()) then return TRUE;
    return FALSE;
```

Library pseudocode for shared/functions/system/InstrSet

```
enumeration InstrSet {InstrSet_A64, InstrSet_A32, InstrSet_T32};
```

Library pseudocode for shared/functions/system/InstructionSynchronizationBarrier

```
InstructionSynchronizationBarrier();
```

Library pseudocode for shared/functions/system/InterruptPending

```
// InterruptPending()
// =====
// Returns TRUE if there are any pending physical or virtual
// interrupts, and FALSE otherwise.

boolean InterruptPending()
    boolean pending_virtual_interrupt = FALSE;
    (irq_pending, -) = IRQPending\(\);
    (fiq_pending, -) = FIQPending\(\);
    boolean pending_physical_interrupt = (irq_pending || fiq_pending ||
                                         IsPhysicalSErrorPending\(\));

    if EL2Enabled\(\) && PSTATE.EL IN {EL0, EL1} && HCR_EL2.TGE == '0' then
        boolean virq_pending = HCR_EL2.IMO == '1' && (VirtualIRQPending\(\) || HCR_EL2.VI == '1') ;
        boolean vfiq_pending = HCR_EL2.FMO == '1' && (VirtualFIQPending\(\) || HCR_EL2.VF == '1');
        boolean vsei_pending = HCR_EL2.AMO == '1' && (IsVirtualSErrorPending\(\) || HCR_EL2.VSE == '1');
        pending_virtual_interrupt = vsei_pending || virq_pending || vfiq_pending;

    return pending_physical_interrupt || pending_virtual_interrupt;
```

Library pseudocode for shared/functions/system/IsASEInstruction

```
// Returns TRUE if the current instruction is an ASIMD or SVE vector instruction.
boolean IsASEInstruction();
```

Library pseudocode for shared/functions/system/IsCMOWControlledInstruction

```
// When using AArch64, returns TRUE if the current instruction is one of IC IVAU,
// DC CIVAC, DC CIGDVAC, or DC CIGVAC.
// When using AArch32, returns TRUE if the current instruction is ICIMVAU or DCCIMVAC.
boolean IsCMOWControlledInstruction();
```

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/IsPhysicalSErrorPending

```
// Returns TRUE if a physical SError interrupt is pending.
boolean IsPhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsSErrorEdgeTriggered

```
// IsSErrorEdgeTriggered()
// =====
// Returns TRUE if the physical SError interrupt is edge-triggered
// and FALSE otherwise.

boolean IsSErrorEdgeTriggered(bits(2) target_el, bits(25) syndrome)
    if HaveRASExt() then
        if HaveDoubleFaultExt() then
            return TRUE;

        if ELUsingAArch32(target_el) then
            if syndrome<11:10> != '00' then
                // AArch32 and not Uncontainable.
                return TRUE;
        else
            if syndrome<24> == '0' && syndrome<5:0> != '000000' then
                // AArch64 and neither IMPLEMENTATION DEFINED syndrome nor Uncategorized.
                return TRUE;
    return boolean IMPLEMENTATION_DEFINED "Edge-triggered SError";
```

Library pseudocode for shared/functions/system/IsSecure

```
// IsSecure()
// =====
// Returns TRUE if current Exception level is in Secure state.

boolean IsSecure()
    if HaveEL(EL3) && !UsingAArch32() && PSTATE.EL == EL3 then
        return TRUE;
    elsif HaveEL(EL3) && UsingAArch32() && PSTATE.M == M32\_Monitor then
        return TRUE;
    return IsSecureBelowEL3();
```

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.
//
// Differs from IsSecure in that it ignores the current EL or Mode
// in considering security state.
// 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_GEN[].NS == '0';
    elseif HaveEL(EL2) && (!HaveSecureEL2Ext() || !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) && HaveSecureEL2Ext() 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/IsSynchronizablePhysicalSErrorPending

```
// Returns TRUE if a synchronizable physical SError interrupt is pending.
boolean IsSynchronizablePhysicalSErrorPending();
```

Library pseudocode for shared/functions/system/IsVirtualSErrorPending

```
// Returns TRUE if a virtual SError interrupt is pending.
boolean IsVirtualSErrorPending();
```

Library pseudocode for shared/functions/system/LocalTimeoutEvent

```
// Returns TRUE if CNTVCT_EL0 equals or exceeds the localtimeout value.
boolean LocalTimeoutEvent(integer localtimeout);
```

Library pseudocode for shared/functions/system/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

```
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

```
enumeration PrivilegeLevel {PL3, PL2, PL1, PL0};
```

Library pseudocode for shared/functions/system/ProcState

```
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) 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 (2) BTYPE,      // Branch Type [v8.5]
  bits (1) ALLINT,     // Interrupt mask bit
  bits (1) SS,         // Software step bit
  bits (1) IL,         // Illegal Execution state bit
  bits (2) EL,         // Exception level
  bits (1) nRW,        // not Register Width: 0=64, 1=32
  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 SCTLR.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/SCRType

```
type SCRType;
```

Library pseudocode for shared/functions/system/SCR_GEN

```
// SCR_GEN[]
// =====

SCRType SCR_GEN[]
// AArch32 secure & AArch64 EL3 registers are not architecturally mapped
assert HaveEL(EL3);
bits(64) r;
if !HaveAArch64() then
    r = ZeroExtend(SCR);
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

```
enumeration SecurityState {
    SS_NonSecure,
    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 !HaveEL(EL3) then
        if SecureOnlyImplementation() then
            return SS_Secure;
        else
            return SS_NonSecure;
    elseif 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

```
// 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

```
// Stores FP Exceptions accumulated by the PE.
SetAccumulatedFPExceptions(bits(8) accumulated_exceptions);
```

Library pseudocode for shared/functions/system/SetPSTATEFromPSR

```
// SetPSTATEFromPSR()
// =====

SetPSTATEFromPSR(bits(N) spsr)
    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;
    boolean from_aarch64 = !UsingAArch32();
    assert N == (if from_aarch64 then 64 else 32);
    PSTATE.SS = DebugExceptionReturnSS(spsr);
    ShouldAdvanceSS = FALSE;
    if illegal_psr_state then
        PSTATE.IL = '1';
        if HaveSSBSEExt() then PSTATE.SSBS = bit UNKNOWN;
        if HaveBTIExt() then PSTATE.BTYPE = bits(2) UNKNOWN;
        if HaveUA0Ext() then PSTATE.UAO = bit UNKNOWN;
        if HaveDITExt() then PSTATE.DIT = bit UNKNOWN;
        if HaveMTEExt() then PSTATE.TCO = bit UNKNOWN;
    else
        // State that is reinstated only on a legal exception return
        PSTATE.IL = spsr<20>;
        if spsr<4> == '1' then // AArch32 state
            AArch32.WriteMode(spsr<4:0>); // Sets PSTATE.EL correctly
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<23>;
        else // AArch64 state
            PSTATE.nRW = '0';
            PSTATE.EL = spsr<3:2>;
            PSTATE.SP = spsr<0>;
            if HaveBTIExt() then PSTATE.BTYPE = spsr<11:10>;
            if HaveSSBSEExt() then PSTATE.SSBS = spsr<12>;
            if HaveUA0Ext() then PSTATE.UAO = spsr<23>;
            if HaveDITExt() then PSTATE.DIT = spsr<24>;
            if HaveMTEExt() then PSTATE.TCO = spsr<25>;

        // 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 HavePANExt() then PSTATE.PAN = spsr<22>;
        if PSTATE.nRW == '1' then // AArch32 state
            PSTATE.Q = spsr<27>;
            PSTATE.IT = RestoredITBits(spsr);
            ShouldAdvanceIT = FALSE;
            if HaveDITExt() 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 HaveFeatNMI() 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/ShouldAdvanceIT

```
boolean ShouldAdvanceIT;
```

Library pseudocode for shared/functions/system/ShouldAdvanceSS

```
boolean ShouldAdvanceSS;
```

Library pseudocode for shared/functions/system/SpeculationBarrier

```
SpeculationBarrier();
```

Library pseudocode for shared/functions/system/SynchronizeContext

```
SynchronizeContext();
```

Library pseudocode for shared/functions/system/SynchronizeErrors

```
// Implements the error synchronization event.  
SynchronizeErrors();
```

Library pseudocode for shared/functions/system/TakeUnmaskedPhysicalSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt.  
TakeUnmaskedPhysicalSErrorInterrupts(boolean iesb_req);
```

Library pseudocode for shared/functions/system/TakeUnmaskedSErrorInterrupts

```
// Take any pending unmasked physical SError interrupt or unmasked virtual SError  
// interrupt.  
TakeUnmaskedSErrorInterrupts();
```

Library pseudocode for shared/functions/system/ThisInstr

```
bits(32) ThisInstr();
```

Library pseudocode for shared/functions/system/ThisInstrLength

```
integer ThisInstrLength();
```

Library pseudocode for shared/functions/system/Unreachable

```
Unreachable()  
    assert FALSE;
```

Library pseudocode for shared/functions/system/UsingAArch32

```
// UsingAArch32()  
// =====  
// Return TRUE if the current Exception level is using AArch32, FALSE if using AArch64.  
  
boolean UsingAArch32()  
    boolean aarch32 = (PSTATE.nRW == '1');  
    if !HaveAArch32() then assert !aarch32;  
    if !HaveAArch64() then assert aarch32;  
    return aarch32;
```

Library pseudocode for shared/functions/system/VirtualFIQPending

```
// Returns TRUE if there is any pending virtual FIQ.  
boolean VirtualFIQPending();
```

Library pseudocode for shared/functions/system/VirtualIRQPending

```
// Returns TRUE if there is any pending virtual IRQ.  
boolean VirtualIRQPending();
```

Library pseudocode for shared/functions/system/WFxType

```
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.  
// - 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 !(IsEventRegisterSet\(\) || (HaveFeatWfxT\(\) && 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.  
// - A Wait for Interrupt with Timeout (WFIT) is executing, and a local timeout event occurs.  
  
WaitForInterrupt(integer localtimeout)  
    if !(HaveFeatWfxT\(\) && LocalTimeoutEvent(localtimeout)) then  
        EnterLowPowerState();  
    return;
```



```
// 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.)

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// The extra argument is used here to allow this example definition. This is an example only and
// does not imply a fixed implementation of these behaviors. Indeed the intention is that it should
// be defined by each implementation, according to its implementation choices.
```

```
Constraint ConstrainUnpredictable(Unpredictable which)
  case which of
    when Unpredictable\_VMSR
      return Constraint\_UNDEF;
    when Unpredictable\_WBOVERLAPLD
      return Constraint\_WBSUPPRESS; // return loaded value
    when Unpredictable\_WBOVERLAPST
      return Constraint\_NONE; // store pre-writeback value
    when Unpredictable\_LDPOVERLAP
      return Constraint\_UNDEF; // instruction is UNDEFINED
    when Unpredictable\_BASEOVERLAP
      return Constraint\_UNKNOWN; // use UNKNOWN address
    when Unpredictable\_DATAOVERLAP
      return Constraint\_UNKNOWN; // store UNKNOWN value
    when Unpredictable\_DEVPAGE2
      return Constraint\_FAULT; // take an alignment fault
    when Unpredictable\_DEVICETAGSTORE
      return Constraint\_NONE; // Do not take a fault
    when Unpredictable\_INSTRDEVICE
      return Constraint\_NONE; // Do not take a fault
    when Unpredictable\_RESCPACR
      return Constraint\_TRUE; // Map to UNKNOWN value
    when Unpredictable\_RESMAIR
      return Constraint\_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable\_S1CTAGGED
      return Constraint\_FALSE; // SCTLX_ELx.C == '0' marks address as untagged
    when Unpredictable\_S2RESMEMATTR
      return Constraint\_NC; // Map to Noncacheable value
    when Unpredictable\_RESTEXCB
      return Constraint\_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable\_RESDACR
      return Constraint\_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable\_RESPRRR
      return Constraint\_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable\_RESVTCRS
      return Constraint\_UNKNOWN; // Map to UNKNOWN value
    when Unpredictable\_RESTnSZ
      return Constraint\_FORCE; // Map to the limit value
    when Unpredictable\_OORTnSZ
      return Constraint\_FORCE; // Map to the limit value
    when Unpredictable\_LARGEIPA
      return Constraint\_FORCE; // Restrict the IA size to the PAMax value
    when Unpredictable\_ESRCONDPASS
      return Constraint\_FALSE; // Report as "AL"
    when Unpredictable\_ILZEROIT
      return Constraint\_FALSE; // Do not zero PSTATE.IT
    when Unpredictable\_ILZEROT
      return Constraint\_FALSE; // Do not zero PSTATE.T
    when Unpredictable\_BPVECTORCATCHPRI
      return Constraint\_TRUE; // Debug Vector Catch: match on 2nd halfword
    when Unpredictable\_VCMATCHHALF
      return Constraint\_FALSE; // No match
    when Unpredictable\_VCMATCHDAPA
      return Constraint\_FALSE; // No match on Data Abort or Prefetch abort
    when Unpredictable\_WPMASKANDBAS
      return Constraint\_FALSE; // Watchpoint disabled
    when Unpredictable\_WPBASCONTIGUOUS
      return Constraint\_FALSE; // Watchpoint disabled
```

```

when Unpredictable\_RESWPMASK
    return Constraint\_DISABLED; // Watchpoint disabled
when Unpredictable\_WPMASKEDBITS
    return Constraint\_FALSE; // Watchpoint disabled
when Unpredictable\_RESBPWPCTRL
    return Constraint\_DISABLED; // Breakpoint/watchpoint disabled
when Unpredictable\_BPNOTIMPL
    return Constraint\_DISABLED; // Breakpoint disabled
when Unpredictable\_RESBPTYPE
    return Constraint\_DISABLED; // Breakpoint disabled
when Unpredictable\_BPNOTCTXCMP
    return Constraint\_DISABLED; // Breakpoint disabled
when Unpredictable\_BPMATCHHALF
    return Constraint\_FALSE; // No match
when Unpredictable\_BPMISMATCHHALF
    return Constraint\_FALSE; // No match
when Unpredictable\_RESTARTALIGNPC
    return Constraint\_FALSE; // Do not force alignment
when Unpredictable\_RESTARTZEROUPPERPC
    return Constraint\_TRUE; // Force zero extension
when Unpredictable\_ZEROUPPER
    return Constraint\_TRUE; // zero top halves of X registers
when Unpredictable\_ERETZEROUPPERPC
    return Constraint\_TRUE; // zero top half of PC
when Unpredictable\_A32FORCEALIGNPC
    return Constraint\_FALSE; // Do not force alignment
when Unpredictable\_SMD
    return Constraint\_UNDEF; // disabled SMC is Unallocated
when Unpredictable\_NVNV1
    return Constraint\_NVNV1\_00; // Map unpredictable configuration of HCR_EL2<NV,NV1>
    // to NV = 0 and NV1 = 0
when Unpredictable\_Shareability
    return Constraint\_OSH; // Map reserved encoding of shareability to Outer Shareable
when Unpredictable\_AFUPDATE
    return Constraint\_TRUE; // AF update for alignment or Permission fault
when Unpredictable\_IESBinDebug // Use SCTLR[].IESB in Debug state
    return Constraint\_TRUE;
when Unpredictable\_BADPMSFCR // Bad settings for PMSFCR_EL1/PMSEVFR_EL1/PMSLATFR_EL1
    return Constraint\_TRUE;
when Unpredictable\_ZEROBTYP
    return Constraint\_TRUE; // Save BTYPE in SPSR_ELx/DPSR_EL0 as '00'
when Unpredictable\_CLEARERRITEZERO // Clearing sticky errors when instruction in flight
    return Constraint\_FALSE;
when Unpredictable\_ALUEXCEPTIONRETURN
    return Constraint\_UNDEF;
when Unpredictable\_DBGxVR\_RESS
    return Constraint\_FALSE;
when Unpredictable\_PMSCR\_PCT
    return Constraint\_PMSCR\_PCT\_VIRT;
when Unpredictable\_WFxTDEBUG
    return Constraint\_FALSE; // WFXT in Debug state does not execute as a NOP
when Unpredictable\_LS64UNSUPPORTED
    return Constraint\_LIMITED\_ATOMICITY; // Accesses are not single-copy atomic above the byte level
// Misaligned exclusives, atomics, acquire/release to region that is not Normal Cacheable WB are a
when Unpredictable\_MISALIGNEDATOMIC
    return Constraint\_FALSE;

when Unpredictable\_IGNORETRAPINDEBUG
    return Constraint\_TRUE; // Trap to register access in debug state is ignored
when Unpredictable\_PMUEVENTCOUNTER
    return Constraint\_UNDEF; // Accesses to the register are UNDEFINED

```

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.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the bits part
// of the result, and may not be applicable in all cases.

(Constraint, bits(width)) ConstrainUnpredictableBits(Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint\_UNKNOWN then
        return (c, Zeros(width)); // See notes; this is an example implementation only
    elsif c == Constraint\_PMSCR\_PCT\_VIRT then
        return (c, Zeros(width));
    else
        return (c, bits(width) UNKNOWN); // bits result not used
```

Library pseudocode for shared/functions/unpredictable/ConstrainUnpredictableBool

```
// ConstrainUnpredictableBool()
// =====

// This is a simple wrapper function for cases where the constrained result is either TRUE or FALSE.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

boolean ConstrainUnpredictableBool(Unpredictable which)

    c = ConstrainUnpredictable(which);
    assert c IN {Constraint\_TRUE, Constraint\_FALSE};
    return (c == Constraint\_TRUE);
```

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.

// NOTE: This version of the function uses an Unpredictable argument to define the call site.
// This argument does not appear in the version used in the Armv8 Architecture Reference Manual.
// See the NOTE on ConstrainUnpredictable() for more information.

// This is an example placeholder only and does not imply a fixed implementation of the integer part
// of the result.

(Constraint, integer) ConstrainUnpredictableInteger(integer low, integer high, Unpredictable which)

    c = ConstrainUnpredictable(which);

    if c == Constraint_UNKNOWN then
        return (c, low);           // See notes; this is an example implementation only
    else
        return (c, integer UNKNOWN); // integer result not used
```

Library pseudocode for shared/functions/unpredictable/Constraint

```
enumeration Constraint    { // General
    Constraint_NONE,           // Instruction executes with
                               // no change or side-effect to its described b
    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
    Constraint_NVNV1_00,
    Constraint_NVNV1_01,
    Constraint_NVNV1_11,
    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_FORCENOSLCHECK,
    // PMSCR_PCT reserved values select Virtual timestamp
    Constraint_PMSCR_PCT_VIRT};
```



```

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_RESPPRR,
    // 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_0ORTnSZ,
    // 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: non-zero MASK and non-ones BAS
    Unpredictable_WPMASKANDBAS,
    // Debug watchpoints: non-contiguous BAS
    Unpredictable_WPBASCONTIGUOUS,
    // Debug watchpoints: reserved MASK
    Unpredictable_RESWPMASK,
    // Debug watchpoints: non-zero 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: 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: 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,
// HCR_EL2.<NV,NV1> == '01'
Unpredictable_NVNV1,
// Reserved shareability encoding
Unpredictable_Shareability,
// Access Flag Update by HW
Unpredictable_AFUPDATE,
// Consider SCTLR[].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,
// WFET or WFIT instruction in Debug state
Unpredictable_WFXTDEBUG,
// Address does not support LS64 instructions
Unpredictable_LS64UNSUPPORTED,
// Misaligned exclusives, atomics, acquire/release to region that is not Normal
Unpredictable_MISALIGNEDATOMIC,
// Clearing DCC/ITR sticky flags when instruction is in flight
Unpredictable_CLEARERRITZ,
// 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 behaviour.
Unpredictable_PMSCR_PCT,

```

```
};
```

Library pseudocode for shared/functions/vector/AdvSIMDEExpandImm

```
// AdvSIMDEExpandImm()
// =====

bits(64) AdvSIMDEExpandImm(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/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
            return PhysicalCountInt() - CNTPOFF_EL2;
        when TimeStamp_None
            return Zeros(64);
        when TimeStamp_CoreSight
            return bits(64) IMPLEMENTATION_DEFINED "CoreSight timestamp";
        otherwise
            Unreachable();
```

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()
    if !\(HaveTraceExt\(\)\) && HaveSelfHostedTrace\(\) then return FALSE;
    if EDSCR.TF0 == '0' then return TRUE;
    if HaveEL\(EL3\) then
        secure_trace_enable = if ELUsingAArch32\(EL3\) then SDCR.STE else MDCR_EL3.STE;
        if secure_trace_enable == '1' && !ExternalSecureNoninvasiveDebugEnabled\(\) then
            return TRUE;
    else
        if SecureOnlyImplementation\(\) && !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;
  ss = SecurityStateAtEL(el);
  if SelfHostedTraceEnabled() then
    boolean trace_allowed;
    // 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';

    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
    case ss of
      when SS_NonSecure return ExternalNoninvasiveDebugEnabled();
      when SS_Secure return ExternalSecureNoninvasiveDebugEnabled();
```

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

```
// Memory barrier instruction that preserves the relative order of memory accesses to System
// registers due to trace operations and other memory accesses to the same registers
TraceSynchronizationBarrier();
```

Library pseudocode for shared/trace/selfhosted/TraceTimeStamp

```
// TraceTimeStamp()
// =====

TimeStamp TraceTimeStamp()
  if SelfHostedTraceEnabled() then
    if HaveEL(EL2) then
      TS_el2 = TRFCR_EL2.TS;
      if !HaveECVExt() && TS_el2 == '10' then
        // Reserved value
        (-, TS_el2) = ConstrainUnpredictableBits(Unpredictable_EL2TIMESTAMP);

      case TS_el2 of
        when '00'
          // Falls out to check TRFCR_EL1.TS
        when '01'
          return TimeStamp_Virtual;
        when '10'
          assert HaveECVExt(); // Otherwise ConstrainUnpredictableBits removes this case
          return TimeStamp_OffsetPhysical;
        when '11'
          return TimeStamp_Physical;

      TS_el1 = TRFCR_EL1.TS;
      if TS_el1 == '00' || (!HaveECVExt() && TS_el1 == '10') then
        // Reserved value
        (-, TS_el1) = ConstrainUnpredictableBits(Unpredictable_EL1TIMESTAMP);

      case TS_el1 of
        when '01'
          return TimeStamp_Virtual;
        when '10'
          assert HaveECVExt();
          return TimeStamp_OffsetPhysical;
        when '11'
          return TimeStamp_Physical;
        otherwise
          Unreachable(); // ConstrainUnpredictableBits removes this case
    else
      return TimeStamp_CoreSight;
```

Library pseudocode for shared/translation/at/ATAccess

```
enumeration ATAccess {
  ATAccess_Read,
  ATAccess_Write,
  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 HaveMTEExt() && memattr.tagged then
    result<7:0> = '11110000';
    return result;

  if memattr.memtype == MemType_Device then
    result<7:4> = '0000';
    if memattr.device == DeviceType_nGnRnE then
      result<3:0> = '0000';
    elsif memattr.device == DeviceType_nGnRE then
      result<3:0> = '0100';
    elsif memattr.device == DeviceType_nGRE then
      result<3:0> = '1000';
    else // DeviceType_GRE
      result<3:0> = '1100';
  else
    if memattr.outer.attr == MemAttr_WT then
      result<7:6> = if memattr.outer.transient then '00' else '10';
      result<5:4> = memattr.outer.hints;
    elsif memattr.outer.attr == 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.attr == MemAttr_WT then
      result<3:2> = if memattr.inner.transient then '00' else '10';
      result<1:0> = memattr.inner.hints;
    elsif memattr.inner.attr == 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 memattr)
  if (memattr.memtype == MemType_Device ||
      (memattr.inner.attr == MemAttr_NC &&
       memattr.outer.attr == MemAttr_NC)) then
    // Force Outer-Shareable on Device and Normal Non-Cacheable memory
    return '10';

  case memattr.shareability of
  when Shareability_NSH return '00';
  when Shareability_ISH return '11';
  when Shareability_OSH return '10';
```

Library pseudocode for shared/translation/at/TranslationStage

```
enumeration TranslationStage {
  TranslationStage_1,
  TranslationStage_12
};
```

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 IN {'x0xx'} then ldfattr.attrs = MemAttr_WT; // Write-through
  elsif attr == '0100' then ldfattr.attrs = MemAttr_NC; // Non-cacheable
  elsif attr IN {'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/MAIRAttr

```
// MAIRAttr()
// =====
// Retrieve the memory attribute encoding indexed in the given MAIR

bits(8) MAIRAttr(integer index, MAIRType mair)
  bit_index = 8 * index;
  return mair<bit_index+7:bit_index>;
```

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.tagged = 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 && HaveFeatXS\(\));
    when '0000xxxx' return attr<1:0> != '00';
    when '01000000' return !(slaarch64 && HaveFeatXS\(\));
    when '10100000' return !(slaarch64 && HaveFeatXS\(\));
    when '11110000' return !(slaarch64 && HaveMTE2Ext\(\));
    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)
  bits(8) attr = attr_in;
  if S1ConstrainUnpredictableRESMAIR(attr, slaarch64) then
    (-, attr) = ConstrainUnpredictableBits(Unpredictable_RESMAIR);

  MemoryAttributes memattrs;
  case attr of
    when '0000xxxx' // Device memory
      memattrs.memtype = MemType\_Device;
      memattrs.device = DecodeDevice(attr<3:2>);
      memattrs.tagged = FALSE;
      memattrs.xs = if slaarch64 then NOT attr<0> else '1';
    when '01000000'
      assert slaarch64 && HaveFeatXS();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = FALSE;
      memattrs.outer.attrs = MemAttr\_NC;
      memattrs.inner.attrs = MemAttr\_NC;
      memattrs.xs = '0';

    when '10100000'
      assert slaarch64 && HaveFeatXS();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = FALSE;
      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 && HaveMTE2Ext();
      memattrs.memtype = MemType\_Normal;
      memattrs.tagged = TRUE;
      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>);
      memattrs.tagged = FALSE;

      if (memattrs.inner.attrs == MemAttr\_WB &&
          memattrs.outer.attrs == MemAttr\_WB) then
        memattrs.xs = '0';
      else
        memattrs.xs = '1';

  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)
    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);

    if ELUsingAArch32(EL2) || !HaveMTE2Ext() then
        memattrs.tagged = FALSE;
    else
        memattrs.tagged = AArch64.IsS2ResultTagged(memattrs, s1_memattrs.tagged);

    memattrs.shareability = S2CombineS1Shareability(s1_memattrs.shareability,
                                                    s2_memattrs.shareability);
    memattrs.xs = s2_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 // Constrained unpredictable
      case ConstrainUnpredictable(Unpredictable_S2RESMEMATTR) of
        when Constraint_NC s2attr.attrs = MemAttr_NC;
        when Constraint_WT s2attr.attrs = MemAttr_WT;
        when Constraint_WB s2attr.attrs = MemAttr_WB;

  // 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

MemoryAttributes S2DecodeMemAttrs(bits(4) attr, bits(2) sh)
  MemoryAttributes memattrs;

  case attr of
    when '00xx' // Device memory
      memattrs.memtype = MemType_Device;
      memattrs.device = DecodeDevice(attr<1:0>);
    otherwise // Normal memory
      memattrs.memtype = MemType_Normal;
      memattrs.outer = S2DecodeCacheability(attr<3:2>);
      memattrs.inner = S2DecodeCacheability(attr<1:0>);

  memattrs.shareability = DecodeShareability(sh);

  return memattrs;
```

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.tagged     = FALSE;
    if (walkmemattrs.inner.attrs == MemAttr\_WB &&
        walkmemattrs.outer.attrs == MemAttr\_WB) then
        walkmemattrs.xs = '0';
    else
        walkmemattrs.xs = '1';

    return walkmemattrs;
```

Library pseudocode for shared/translation/faults/AlignmentFault

```
// AlignmentFault()
// =====

FaultRecord AlignmentFault(AccType acctype, boolean iswrite, boolean secondstage)
    FaultRecord fault;

    fault.statuscode = Fault\_Alignment;
    fault.acctype    = acctype;
    fault.write      = iswrite;
    fault.secondstage = secondstage;

    return fault;
```

Library pseudocode for shared/translation/faults/AsyncExternalAbort

```
// AsyncExternalAbort()
// =====
// Return a fault record indicating an asynchronous External abort

FaultRecord AsyncExternalAbort(boolean parity, bits(2) errortype, bit extflag)
    FaultRecord fault;

    fault.statuscode = if parity then Fault\_AsyncParity else Fault\_AsyncExternal;
    fault.extflag    = extflag;
    fault.errortype  = errortype;
    fault.acctype    = AccType\_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk = FALSE;

    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.statuscode = Fault_None;
    fault.acctype    = AccType_NORMAL;
    fault.secondstage = FALSE;
    fault.s2fslwalk  = FALSE;

    return fault;
```

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 HaveVirtHostExt() && 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[], etc.) implicitly
// return the correct value.

bits(2) S1TranslationRegime()
    return S1TranslationRegime(PSTATE.EL);
```

Library pseudocode for shared/translation/vmsa/AddressDescriptor

```
type AddressDescriptor is (
    FaultRecord    fault,          // fault.statuscode indicates whether the address is valid
    MemoryAttributes memattrs,
    FullAddress    paddress,
    bits(64)       vaddress
)

constant integer FINAL_LEVEL = 3;
```

Library pseudocode for shared/translation/vmsa/ContiguousSize

```
// ContiguousSize()
// =====
// Return the number of entries log 2 marking a contiguous output range

integer ContiguousSize(TGx tgx, integer level)
  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.address = pa;
  addrdesc.vaddress = va;
  addrdesc.memattrs = memattrs;
  addrdesc.fault = NoFault();

  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/DescriptorType

```
enumeration DescriptorType {
  DescriptorType_Table,
  DescriptorType_Block,
  DescriptorType_Page,
  DescriptorType_Invalid
};
```

Library pseudocode for shared/translation/vmsa/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, FaultRecord fault_in)
// 32-bit descriptors for AArch32 Short-descriptor format
// 64-bit descriptors for AArch64 or AArch32 Long-descriptor format
assert N == 32 || N == 64;
bits(N) descriptor;
FaultRecord fault = fault_in;
AccessDescriptor walkacc;

walkacc.acctype = AccType_TTW;
// MPAM PARTID for translation table walk is determined by the access invoking the translation
walkacc.mpam = GenMPAMcurEL(fault.acctype);

PhysMemRetStatus memstatus;
(memstatus, descriptor) = PhysMemRead(walkaddress, N DIV 8, walkacc);
if IsFault(memstatus) then
    fault = HandleExternalTTWAbort(memstatus, fault.write, walkaddress,
                                   walkacc, 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/IsAtomicRW

```
// IsAtomicRW()
// =====
// Is the access an atomic operation?

boolean IsAtomicRW(AccType acctype)
return acctype IN {
    AccType_ATOMICRW,
    AccType_ORDEREDRW,
    AccType_ORDEREDATOMICRW
};
```

Library pseudocode for shared/translation/vmsa/Regime

```
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

```
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      e0pd,        // TCR_EL1.E0PDx or TCR_EL2.E0PDx when HCR_EL2.E2H == '1'
    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'

// 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 HPCR.IRGN0
    bits(2)  orgn,        // TCR_ELx.ORGNx / TTBCR.ORGnx or HPCR.ORGn0
    bits(2)  sh,          // TCR_ELx.SHx / TTBCR.SHx or HPCR.SH0
    bit      hpd,         // TCR_ELx.HPD{x} / TTBCR2.HPDx or HPCR.HPD
    bit      ee,          // SCTLr_ELx.EE / SCTLr.EE or HSCTLr.EE
    bit      wxn,         // SCTLr_ELx.WXN / SCTLr.WXN or HSCTLr.WXN
    bit      ntlsmid,     // SCTLr_ELx.nTlSMID / SCTLr.nTlSMID or HSCTLr.nTlSMID
    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

```
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      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      fwb,     // HCR_EL2.PTW  
  bit      cmow,    // HCRX_EL2.CMOW  
  
// 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/SDFTType

```
enumeration SDFTType {  
  SDFTType_Table,  
  SDFTType_Invalid,  
  SDFTType_Supersection,  
  SDFTType_Section,  
  SDFTType_LargePage,  
  SDFTType_SmallPage  
};
```

Library pseudocode for shared/translation/vmsa/SecurityStateForRegime

```
// SecurityStateForRegime()  
// =====  
// Return the Security State of the given translation regime  
  
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, TGx tgx, ITWState walkstate)
// Output Address
FullAddress oa;
integer csize;

tsize = TranslationSize(tgx, walkstate.level);
if walkstate.contiguous == '1' then
    csize = ContiguousSize(tgx, walkstate.level);
else
    csize = 0;

ia_msb = tsize + csize;
oa.paspace = walkstate.baseaddress.paspace;
oa.address = walkstate.baseaddress.address<51:ia_msb>:ia<ia_msb-1:0>;

return oa;
```

Library pseudocode for shared/translation/vmsa/TGx

```
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

integer 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

```
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,
    bits(64) ia, // Input Address
    TGx tg,
    bit cnp,
    bit xs // XS attribute (FEAT_XS)
)
```

Library pseudocode for shared/translation/vmsa/TLBRecord

```
type TLBRecord is (  
  TLBContext context,  
  TTWState walkstate,  
  integer blocksize, // Number of bits directly mapped from IA to OA  
  integer contigsize, // Number of entries log 2 marking a contiguous output range  
  bits(64) s1descriptor, // Stage 1 leaf descriptor in memory (valid if the TLB caches stage 1)  
  bits(64) s2descriptor // Stage 2 leaf descriptor in memory (valid if the TLB caches stage 2)  
)
```

Library pseudocode for shared/translation/vmsa/TTWState

```
type TTWState is (  
  boolean istable,  
  integer level,  
  FullAddress baseaddress,  
  bit contiguous,  
  bit nG,  
  bit guardedpage,  
  SDFType sdfstype, // 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_in, AccType acctype)  
  
  bits(2) el;  
  case acctype of  
    when AccType\_NV2REGISTER  
      assert EL2Enabled\(\) && el_in == EL1;  
      el = EL2;  
    otherwise  
      el = el_in;  
  
  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

integer TranslationSize(TGx tgx, integer level)
    granulebits = TGxGranuleBits(tgx);
    blockbits   = (FINAL_LEVEL - level) * (granulebits - 3);

    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 access)
    return HasUnprivileged(access.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(TLBContext access)
    return access.regime == Regime_EL10 && EL2Enabled();
```

Library pseudocode for shared/translation/vmsa/VARange

```
enumeration VARange {
    VARange_LOWER,
    VARange_UPPER
};
```

Internal version only: isa v01_27, pseudocode v2022-03_rel ; Build timestamp: 2022-03-29T10:46
Copyright © 2010-2022 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.