

(old)

htmldiff from-

(new)

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.


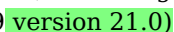
~~If any of the provisions contained in these terms conflict with any of the provisions of any click through or signed written agreement covering this document with Arm, then the click through or signed written agreement prevails over and supersedes the conflicting provisions of these terms.~~ This document 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.


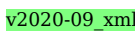
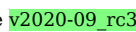
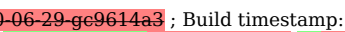
The Arm corporate logo and words marked with ® or  are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. Please follow Arm's trademark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

Copyright © 2020 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)

Internal version only: isa  v01_19v01_15, pseudocode  v2020-09_xmlv2020-06_rel, sve  v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp:  2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

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).

SMLSD, SMLSDX: Signed Multiply Subtract Dual.

SMLSLD, SMLSLDX: Signed Multiply Subtract Long Dual.

SMMLA, SMMLAR: Signed Most Significant Word Multiply Accumulate.

SMMLS, SMMLSR: Signed Most Significant Word Multiply Subtract.

SMMUL, SMMULR: Signed Most Significant Word Multiply.

SMUAD, SMUADX: Signed Dual Multiply Add.

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

SMULL, SMULLS: Signed Multiply Long.

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

SMUSD, SMUSDX: Signed Multiply Subtract Dual.

SRS, SRSDA, SRSDB, SRSIA, SRSIB: Store Return State.

SSAT: Signed Saturate.

SSAT16: Signed Saturate 16.

SSAX: Signed Subtract and Add with Exchange.

SSBB: Speculative Store Bypass Barrier.

SSUB16: Signed Subtract 16.

SSUB8: Signed Subtract 8.

STC: Store data to System register.

STL: Store-Release Word.

STLB: Store-Release Byte.

STLEX: Store-Release Exclusive Word.

STLEXB: Store-Release Exclusive Byte.

STLEXD: Store-Release Exclusive Doubleword.

STLEXH: Store-Release Exclusive Halfword.

STLH: Store-Release Halfword.

STM (User registers): Store Multiple (User registers).

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

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_19](#)~~v01-15~~, pseudocode [v2020-09_xml](#)~~v2020-06_rel~~, sve [v2020-09_rc3](#)~~v2020-06-29-gc9614a3~~; Build timestamp: [2020-09-30T21:35:36](#)~~2020-07-03T11:35:36~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 modeCPS (imod == 00 && M == 1)

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

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

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

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

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

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

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

Interrupt enable and change modeCPSIE (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 disableCPSID (im == 1)

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

Interrupt enableCPSIE (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 modeCPS (imod == 00 && M == 1)

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

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

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

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

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

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

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

Interrupt enable and change modeCPSIE (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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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.

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
!=	1	1	1	1	0	0	1	1	0	0	1	0	0	0	0	(1)	(1)	(1)	(1)	(0)	(0)	(0)	(0)	0	0	0	1	0	1	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>}{<w>}

```
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_19v01-15~~, pseudocode ~~v2020-09_xmlv2020-06-rel~~, sve ~~v2020-09_rc3v2020-06-29-gc9614a3~~ ; Build timestamp: ~~2020-09-30T21:20:07+00:00~~ ~~2020-07-03T11:35:36~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 this instruction, 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 = SCTLR.EE;
        if HavePANExt() && SCTLR.SPAN == '0' then PSTATE.PAN = '1';
        LR_svc = bits(32) UNKNOWN;
        SPSR_svc = bits(32) UNKNOWN;
    else
        PSTATE.E = HSCTLR.EE;
        ELR_hyp = bits(32) UNKNOWN;
        HSR = bits(32) UNKNOWN;
        SPSR_hyp = bits(32) UNKNOWN;

    DLR = bits(32) UNKNOWN;
    DSPSR = bits(32) UNKNOWN;
else // Targeting EL1 using AArch64
    AArch64.MaybeZeroRegisterUppers();
    MaybeZeroSVEUppers(EL1);
    PSTATE.nRW = '0';
    PSTATE.SP = '1';
    PSTATE.EL = EL1;
    if HavePANExt() && SCTLR_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;
    ESR_EL1 = bits(32) UNKNOWN;
    SPSR_EL1 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

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

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

Internal version only: isa ~~v01_19~~~~v01_15~~, pseudocode ~~v2020-09_xml~~~~v2020-06_rel~~, sve ~~v2020-09_rc3~~~~v2020-06-29-gc9614a3~~; Build timestamp: ~~2020-09-30T21:20:07+00:00~~~~2020-07-03T11:35:36~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 this instruction, see [DCPS](#).

T1

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

T1

DCPS2

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

Operation

```
if !Halted() || IsSecure() 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 HaveUAOExt() then PSTATE.UAO = '0';

    ELR_EL2 = bits(64) UNKNOWN;
    ESR_EL2 = bits(64) UNKNOWN;
    SPSR_EL2 = bits(64) UNKNOWN;
    ESR_EL2 = bits(32) UNKNOWN;
    SPSR_EL2 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) 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
```

Internal version only: isa ~~v01_19~~~~v01_15~~, pseudocode ~~v2020-09_xml~~~~v2020-06_rel~~, sve ~~v2020-09_rc3~~~~v2020-06-29-gc9614a3~~; Build timestamp: ~~2020-09-30T21:2020-07-03T11:3536~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 this instruction, see [DCPS](#).

T1

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

T1

DCPS3

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

Operation

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

if ELUsingAArch32(EL3) then
    from_secure = IsSecure();
    if PSTATE.M == M32_Monitor then SCR.NS = '0';
    AArch32.WriteMode(M32_Monitor);
    if HavePANExt() then
        if !from_secure then
            PSTATE.PAN = '0';
        elsif SCTL.R.SPAN == '0' then
            PSTATE.PAN = '1';
    PSTATE.E = SCTL.R.EE;

    LR_mon = bits(32) UNKNOWN;
    SP_S_R_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.UAO = '0';

    ELR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(64) UNKNOWN;
    SPSR_EL3 = bits(64) UNKNOWN;
    ESR_EL3 = bits(32) UNKNOWN;
    SPSR_EL3 = bits(32) UNKNOWN;

    DLR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(64) UNKNOWN;
    DSPSR_EL0 = bits(32) UNKNOWN;

    sync_errors = HaveIESB() && SCTL.R.EL3.IESB == '1';
    if HaveDoubleFaultExt() && SCR_EL3.EA == '1' && SCR_EL3.NMEA == '1' then
        sync_errors = TRUE;
    // SCTL.R.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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

ERET

Exception Return.

The PE branches to the address held in the register holding the preferred return address, and restores *PSTATE* from *SPSR_<current_mode>*.

The register holding the preferred return address is:

- *ELR_hyp*, when executing in Hyp mode.
- LR, when executing in a mode other than Hyp mode, User mode, or System mode.

The PE checks *SPSR_<current_mode>* for an illegal return event. See *Illegal return events from AArch32 state*.

Exception Return is CONSTRAINED UNPREDICTABLE in User mode and System mode.

In Debug state, the T1 encoding of ERET executes the DRPS operation.

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

A1

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

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 // Perform DRPS operation in Debug state
    if PSTATE.M == M32_User then
        UNDEFINED;
    elseif PSTATE.M == M32_System then
        UNPREDICTABLE; // UNDEFINED or NOP
    else
        SynchronizeContext();
        bits(32) spsr = (); SetPSTATEFromPSR(SPSR[];
        SetPSTATEFromPSR(spsr);
[]);

// PSTATE.{N,Z,C,V,Q,GE,SS,A,I,F} are not observable and ignored in Debug state, so
// behave as if UNKNOWN.
PSTATE.<N,Z,C,V,Q,GE,SS,A,I,F> = bits(13) UNKNOWN;
// In AArch32 Debug state, all instructions are T32 and unconditional.
PSTATE.IT = '00000000'; PSTATE.T = '1'; // PSTATE.J is RES0
DLR = bits(32) UNKNOWN; DSPSR = bits(32) UNKNOWN;
UpdateEDSCRFields(); // Update EDSCR PE state flags
```

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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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

(Armv8.2)

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

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

(Armv8.2)

15	14	13	12	11	10	9	8	7	6	5	4	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>}{~~{}.~~W<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.ESB0operation();
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() then AArch32.vESB0operation();
    TakeUnmaskedSErrorInterrupts();
```

Internal version only: isa [v01_19v01_15](#), pseudocode [v2020-09_xmlv2020-06_rel](#), sve [v2020-09_rc3v2020-06-29-ge9614a3](#); Build timestamp: [2020-09-30T21:2020-07-03T11:3536](#)

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 = MemQ[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(() then value<63:32> else value<31:0>; AccType_ORDERED) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType_ORDERED) then value<31:0> else value<63:32>; () 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_19~~v01_15, pseudocode ~~v2020-09_xml~~v2020-06_rel, sve ~~v2020-09_rc3~~v2020-06-29-gc9614a3; Build timestamp: ~~2020-09-30T21:35:36~~2020-07-03T11:35:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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

15	14	13	12	11	10	9	8	7	6	5	4	3	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															
Rn																Rt				Rt2				imm8							

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(()) then 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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() then 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 <i>Standard assembler syntax fields</i> . |
| <q> | See <i>Standard assembler syntax fields</i> . |
| <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() then 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:00Z2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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(()) then value<63:32> else value<31:0>; AccType_ATOMIC) then value<63:32> else value<31:0>;
    R[t2] = if BigEndian(AccType_ATOMIC) then value<31:0> else value<63:32>; (()) 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 00001111 00000011 11011111';
    if HavePANExt() then
        mask<22> = '1';

    if HaveDITExt() then
        mask<21> = '1';
    psr_val = GetPSRFromPSTATE(()) AND mask;
    if PSTATE.EL == 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

MRS (Banked register)

Move to Register from Banked or Special register moves the value from the Banked general-purpose register or *Saved Program Status Registers (SPSRs)* of the specified mode, or the value of *ELR_hyp*, to a general-purpose register.

MRS (Banked register) is UNPREDICTABLE if executed in User mode.

When EL3 is using AArch64, if an MRS (Banked register) instruction that is executed in a Secure EL1 mode would access SPSR_mon, SP_mon, or LR_mon, it is trapped to EL3.

The effect of using an MRS (Banked register) instruction with a register argument that is not valid for the current mode is UNPREDICTABLE. For more information see *Usage restrictions on the Banked register transfer instructions*.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111				0	0	0	1	0	R	0	0	M1				Rd				(0)	(0)	1	M	0	0	0	0	(0)	(0)	(0)	(0)
cond																															

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)

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>;
[d] = SPSR_fiq;
                when '10000' R[d] = SPSR_irq<31:0>;
[d] = SPSR_irq;
                when '10010' R[d] = SPSR_svc<31:0>;
[d] = SPSR_svc;
                when '10100' R[d] = SPSR_abt<31:0>;
[d] = SPSR_abt;
                when '10110' R[d] = SPSR_und<31:0>;
[d] = SPSR_und;
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    R[d] = SPSR_mon;
                when '11110' R[d] = SPSR_hyp<31:0>;
[d] = SPSR_hyp;
            else
                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 v01.19v01.15, pseudocode v2020-09.xmlv2020-06-rel, sve v2020-09.rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 '10110' SPSR_und = R[n];
                when '11100'
                    if !ELUsingAArch32(EL3) then AArch64.MonitorModeTrap();
                    SPSR_mon = R[n];
                when '11110' SPSR_hyp = R[n];
            else
                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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 *PSSTATE*. This word must be in the format of an SPSR.

An address adjusted by the size of the data loaded can optionally be written back to the base register.

The PE checks the value of the word loaded from the higher address for an illegal return event. See *Illegal return events from AArch32 state*.

RFE is UNDEFINED in Hyp mode and CONSTRAINED UNPREDICTABLE in User mode.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	P	U	0	W	1		Rn			(0)	(0)	(0)	(0)	(1)	(0)	(1)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)

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)

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

Increment After (P == 0 && U == 1)

RFEIA{<c>}{<q>} <Rn>{!} // (Preferred syntax)

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

Increment Before (P == 1 && U == 1)

RFEIB{<c>}{<q>} <Rn>{!} // (Preferred syntax)

RFEIBA{<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)	(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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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(() then 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 = (U == '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() then
            data<63:32> = 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<Rm>	Is the general-purpose index register, encoded in the "Rm" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    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() then
            data<63:32> = 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_19v01_15`, pseudocode `v2020-09_xmlv2020-06_rel`, sve `v2020-09_rc3v2020-06-29-gc9614a3`; Build timestamp: `2020-09-30T21:2020-07-03T11:3536`

Copyright © 2010-2020 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(()) then 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

TSB CSYNC

Trace Synchronization Barrier. This instruction is a barrier that synchronizes the trace operations of instructions. If ~~FEAT_SHTrace is not implemented, this instruction executes as a~~ *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

(Armv8.4)

31	30	29	28	27	26	25	24	23	22	21	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	(1)	(1)	(1)	(1)	(0)	(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;           // 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

(Armv8.4)

15	14	13	12	11	10	9	8	7	6	5	4	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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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	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	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 which may be EL1 or EL2.
        AArch32.CheckForWfxTrap(EL1, TRUE);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch32.CheckForWfxTrap(EL2, TRUE);
    if HaveEL(EL3) && PSTATE.M != M32_Monitor then
        // Check for traps described by the Secure Monitor.
        AArch32.CheckForWfxTrap(EL3, TRUE);
integer localtimeout = -1; // No local timeout event is generated, TRUE);
WaitForEvent(localtimeout);;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 which may be EL1 or EL2.
        AArch32.CheckForWfxTrap(EL1, FALSE);
    if PSTATE.EL IN {EL0, EL1} && EL2Enabled() && !IsInHost() then
        // Check for traps described by the Hypervisor.
        AArch32.CheckForWfxTrap(EL2, FALSE);
    if HaveEL(EL3) && PSTATE.M != M32_Monitor then
        // Check for traps described by the Secure Monitor.
        AArch32.CheckForWfxTrap(EL3, FALSE);
integer localtimeout = -1; // No local timeout event is generated, FALSE);
WaitForInterrupt(localtimeout);();
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

(old)

htmldiff from-

(new)

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.

[VCVTI](#): Convert to or from a half-precision value in the top half of a single-precision register.

[VCVTT \(BFloat16\)](#): Converts from a single-precision value to a BFloat16 value in the top half of a single-precision register..

[VDIV](#): Divide.

VDOT (by element): BFloat16 floating-point indexed dot product (vector, by element).

VDOT (vector): BFloat16 floating-point (BF16) dot product (vector).

VDUP (general-purpose register): Duplicate general-purpose register to vector.

VDUP (scalar): Duplicate vector element to vector.

VEOR: Vector Bitwise Exclusive OR.

VEXT (byte elements): Vector Extract.

VEXT (multibyte elements): Vector Extract: an alias of VEXT (byte elements).

[VFMA](#): Vector Fused Multiply Accumulate.

VFMA, VFMA, VFMA (BFloat16, by scalar): BFloat16 floating-point widening multiply-add long (by scalar).

VFMA, VFMA, VFMA (BFloat16, vector): BFloat16 floating-point widening multiply-add long (vector).

VFMA, VFMA (by scalar): Vector Floating-point Multiply-Add Long to accumulator (by scalar).

VFMA, VFMA (vector): Vector Floating-point Multiply-Add Long to accumulator (vector).

[VFMS](#): Vector Fused Multiply Subtract.

VFMS, VFMS (by scalar): Vector Floating-point Multiply-Subtract Long from accumulator (by scalar).

VFMS, VFMS (vector): Vector Floating-point Multiply-Subtract Long from accumulator (vector).

[VFNMA](#): Vector Fused Negate Multiply Accumulate.

[VFNMS](#): Vector Fused Negate Multiply Subtract.

VHADD: Vector Halving Add.

VHSUB: Vector Halving Subtract.

VINS: Vector move Insertion.

[VJCVT](#): Javascript Convert to signed fixed-point, rounding toward Zero.

[VLD1 \(multiple single elements\)](#): Load multiple single 1-element structures to one, two, three, or four registers.

VLD1 (single element to all lanes): Load single 1-element structure and replicate to all lanes of one register.

VLD1 (single element to one lane): Load single 1-element structure to one lane of one register.

VLD2 (multiple 2-element structures): Load multiple 2-element structures to two or four registers.

VLD2 (single 2-element structure to all lanes): Load single 2-element structure and replicate to all lanes of two registers.

VLD2 (single 2-element structure to one lane): Load single 2-element structure to one lane of two registers.

VLD3 (multiple 3-element structures): Load multiple 3-element structures to three registers.

VLD3 (single 3-element structure to all lanes): Load single 3-element structure and replicate to all lanes of three registers.

VLD3 (single 3-element structure to one lane): Load single 3-element structure to one lane of three registers.

VLD4 (multiple 4-element structures): Load multiple 4-element structures to four registers.

VLD4 (single 4-element structure to all lanes): Load single 4-element structure and replicate to all lanes of four registers.

VLD4 (single 4-element structure to one lane): Load single 4-element structure to one lane of four registers.

[VLDM, VLDMDB, VLDMIA](#): Load Multiple SIMD&FP registers.

[VLDR \(immediate\)](#): Load SIMD&FP register (immediate).

[VLDR \(literal\)](#): Load SIMD&FP register (literal).

VMAX (floating-point): Vector Maximum (floating-point).

VMAX (integer): Vector Maximum (integer).

[VMAXNM](#): Floating-point Maximum Number.

VMIN (floating-point): Vector Minimum (floating-point).

VMIN (integer): Vector Minimum (integer).

[VMINNM](#): Floating-point Minimum Number.

VMLA (by scalar): Vector Multiply Accumulate (by scalar).

[VMLA \(floating-point\)](#): Vector Multiply Accumulate (floating-point).

VMLA (integer): Vector Multiply Accumulate (integer).

VMLAL (by scalar): Vector Multiply Accumulate Long (by scalar).

VMLAL (integer): Vector Multiply Accumulate Long (integer).

VMLS (by scalar): Vector Multiply Subtract (by scalar).

[VMLS \(floating-point\)](#): Vector Multiply Subtract (floating-point).

VMLS (integer): Vector Multiply Subtract (integer).

VMLSL (by scalar): Vector Multiply Subtract Long (by scalar).

VMLSL (integer): Vector Multiply Subtract Long (integer).

VMMLA: BFloat16 floating-point matrix multiply-accumulate.

VMOV (between general-purpose register and half-precision): Copy 16 bits of a general-purpose register to or from a 32-bit SIMD&FP register.

VMOV (between general-purpose register and single-precision): Copy a general-purpose register to or from a 32-bit SIMD&FP register.

VMOV (between two general-purpose registers and a doubleword floating-point register): Copy two general-purpose registers to or from a SIMD&FP register.

VMOV (between two general-purpose registers and two single-precision registers): Copy two general-purpose registers to a pair of 32-bit SIMD&FP registers.

VMOV (general-purpose register to scalar): Copy a general-purpose register to a vector element.

VMOV (immediate): Copy immediate value to a SIMD&FP register.

VMOV (register): Copy between FP registers.

VMOV (register, SIMD): Copy between SIMD registers: an alias of VORR (register).

VMOV (scalar to general-purpose register): Copy a vector element to a general-purpose register with sign or zero extension.

VMOVL: Vector Move Long.

VMOVN: Vector Move and Narrow.

VMOVX: Vector Move extraction.

VMRS: Move SIMD&FP Special register to general-purpose register.

VMSR: Move general-purpose register to SIMD&FP Special register.

VMUL (by scalar): Vector Multiply (by scalar).

[VMUL \(floating-point\)](#): Vector Multiply (floating-point).

VMUL (integer and polynomial): Vector Multiply (integer and polynomial).

VMULL (by scalar): Vector Multiply Long (by scalar).

VMULL (integer and polynomial): Vector Multiply Long (integer and polynomial).

VMVN (immediate): Vector Bitwise NOT (immediate).

VMVN (register): Vector Bitwise NOT (register).

VNEG: Vector Negate.

[VNMLA](#): Vector Negate Multiply Accumulate.

[VNMLS](#): Vector Negate Multiply Subtract.

[VNMUL](#): Vector Negate Multiply.

VORN (immediate): Vector Bitwise OR NOT (immediate): an alias of VORR (immediate).

VORN (register): Vector bitwise OR NOT (register).

VORR (immediate): Vector Bitwise OR (immediate).

VORR (register): Vector bitwise OR (register).

VPADAL: Vector Pairwise Add and Accumulate Long.

VPADD (floating-point): Vector Pairwise Add (floating-point).

VPADD (integer): Vector Pairwise Add (integer).

VPADDL: Vector Pairwise Add Long.

VPMAX (floating-point): Vector Pairwise Maximum (floating-point).

VPMAX (integer): Vector Pairwise Maximum (integer).

VPMIN (floating-point): Vector Pairwise Minimum (floating-point).

VPMIN (integer): Vector Pairwise Minimum (integer).

VPOP: Pop SIMD&FP registers from Stack: an alias of VLDM, VLDMDB, VLDMIA.

VPUSH: Push SIMD&FP registers to Stack: an alias of VSTM, VSTMDB, VSTMIA.

VQABS: Vector Saturating Absolute.

VQADD: Vector Saturating Add.

VQDMLAL: Vector Saturating Doubling Multiply Accumulate Long.

VQDMLSL: Vector Saturating Doubling Multiply Subtract Long.

VQDMULH: Vector Saturating Doubling Multiply Returning High Half.

VQDMULL: Vector Saturating Doubling Multiply Long.

VQMOVN, VQMOVUN: Vector Saturating Move and Narrow.

VQNEG: Vector Saturating Negate.

VQRDMLAH: Vector Saturating Rounding Doubling Multiply Accumulate Returning High Half.

VQRDMLSH: Vector Saturating Rounding Doubling Multiply Subtract Returning High Half.

VQRDMULH: Vector Saturating Rounding Doubling Multiply Returning High Half.

VQRSHL: Vector Saturating Rounding Shift Left.

VQRSHRN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQRSHRN, VQRSHRUN: Vector Saturating Rounding Shift Right, Narrow.

VQRSHRUN (zero): Vector Saturating Rounding Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHL (register): Vector Saturating Shift Left (register).

VQSHL, VQSHLU (immediate): Vector Saturating Shift Left (immediate).

VQSHRN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSHRN, VQSHRUN: Vector Saturating Shift Right, Narrow.

VQSHRUN (zero): Vector Saturating Shift Right, Narrow: an alias of VQMOVN, VQMOVUN.

VQSUB: Vector Saturating Subtract.

VRADDHN: Vector Rounding Add and Narrow, returning High Half.

[VRECPE](#): Vector Reciprocal Estimate.

VRECPS: Vector Reciprocal Step.

VREV16: Vector Reverse in halfwords.

VREV32: Vector Reverse in words.

VREV64: Vector Reverse in doublewords.

VRHADD: Vector Rounding Halving Add.

VRINTA (Advanced SIMD): Vector Round floating-point to integer towards Nearest with Ties to Away.

[VRINTA \(floating-point\)](#): Round floating-point to integer to Nearest with Ties to Away.

VRINTM (Advanced SIMD): Vector Round floating-point to integer towards -Infinity.

[VRINTM \(floating-point\)](#): Round floating-point to integer towards -Infinity.

VRINTN (Advanced SIMD): Vector Round floating-point to integer to Nearest.

[VRINTN \(floating-point\)](#): Round floating-point to integer to Nearest.

VRINTP (Advanced SIMD): Vector Round floating-point to integer towards +Infinity.

[VRINTP \(floating-point\)](#): Round floating-point to integer towards +Infinity.

[VRINTR](#): Round floating-point to integer.

VRINTX (Advanced SIMD): Vector round floating-point to integer inexact.

[VRINTX \(floating-point\)](#): Round floating-point to integer inexact.

VRINTZ (Advanced SIMD): Vector round floating-point to integer towards Zero.

[VRINTZ \(floating-point\)](#): Round floating-point to integer towards Zero.

VRSHL: Vector Rounding Shift Left.

VRSHR: Vector Rounding Shift Right.

VRSHR (zero): Vector Rounding Shift Right: an alias of VORR (register).

VRSHRN: Vector Rounding Shift Right and Narrow.

VRSHRN (zero): Vector Rounding Shift Right and Narrow: an alias of VMOVN.

[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.

VSRA: Vector Shift Right and Accumulate.

VSRI: Vector Shift Right and Insert.

[VST1 \(multiple single elements\)](#): Store multiple single elements from one, two, three, or four registers.

VST1 (single element from one lane): Store single element from one lane of one register.

VST2 (multiple 2-element structures): Store multiple 2-element structures from two or four registers.

VST2 (single 2-element structure from one lane): Store single 2-element structure from one lane of two registers.

VST3 (multiple 3-element structures): Store multiple 3-element structures from three registers.

VST3 (single 3-element structure from one lane): Store single 3-element structure from one lane of three registers.

VST4 (multiple 4-element structures): Store multiple 4-element structures from four registers.

VST4 (single 4-element structure from one lane): Store single 4-element structure from one lane of four registers.

[VSTM](#), [VSTMDB](#), [VSTMIA](#): Store multiple SIMD&FP registers.

VSTR: Store SIMD&FP register.

VSUB (floating-point): Vector Subtract (floating-point).

VSUB (integer): Vector Subtract (integer).

VSUBHN: Vector Subtract and Narrow, returning High Half.

VSUBL: Vector Subtract Long.

VSUBW: Vector Subtract Wide.

VSUDOT (by element): Dot Product index form with signed and unsigned integers (by element).

VSWP: Vector Swap.

VTBL, VTBX: Vector Table Lookup and Extension.

VTRN: Vector Transpose.

VTST: Vector Test Bits.

VUDOT (by element): Dot Product index form with unsigned integers..

VUDOT (vector): Dot Product vector form with unsigned integers..

VUMMLA: Widening 8-bit unsigned integer matrix multiply-accumulate into 2x2 matrix.

VUSDOT (by element): Dot Product index form with unsigned and signed integers (by element).

VUSDOT (vector): Dot Product vector form with mixed-sign integers.

VUSMMLA: Widening 8-bit mixed integer matrix multiply-accumulate into 2x2 matrix.

VUZP: Vector Unzip.

VUZP (alias): Vector Unzip: an alias of VTRN.

VZIP: Vector Zip.

VZIP (alias): Vector Zip: an alias of VTRN.

Internal version only: isa **v01_19v01_15**, pseudocode **v2020-09_xmlv2020-06_rel**, sve **v2020-09_rc3v2020-06-29-ge9614a3**; Build timestamp: **2020-09-30T21:2020-07-03T11:3536**

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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(()) then word1:word2 else word2:word1;
if wback then AccType_ATOMIC) then word1:word2 else word2:word1;
if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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(()) then AccType_ATOMIC then D[d+r]<63:32> else D[d+r]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then(()) 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;
```

Internal version only: isa v01_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07+00:003536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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;
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) (Armv8.2)

```
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;
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;
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)
(Armv8.2)

```
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;
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 <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding A2, T1 and T2: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<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] = FAdd(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) : FAdd(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            when 32
                S[d] = FAdd(S[n], S[m], FPSCR[]);
            when 64
                D[d] = FAdd(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 [HCPTR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to 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)

(Armv8.2)

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;
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) (Armv8.2)

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;
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) (Armv8.2)

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>

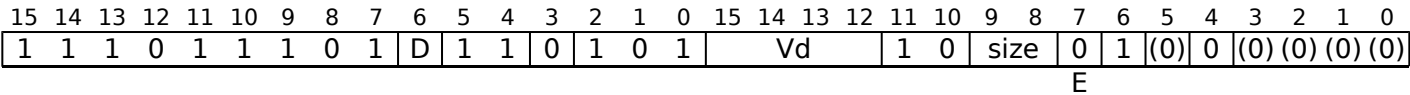
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
quiet_nan_exc = (E == '1'); with_zero = FALSE;
case size of
  when '01' esize = 16; d = UInt(Vd:D); m = UInt(Vm:M);
  when '10' esize = 32; d = UInt(Vd:D); m = UInt(Vm:M);
  when '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



Half-precision scalar (size == 01)
(Armv8.2)

```
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;
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[]);
            [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[]);
            [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[]);
            [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. 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_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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)

(Armv8.2)

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;
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)
cond																E															

Half-precision scalar (size == 01) (Armv8.2)

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;
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) (Armv8.2)

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;
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) (Armv8.2)

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;
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[]);
            [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[]);
            [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[]);
            [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. 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_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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(); CheckVFPEEnabled(TRUE);
    if double_to_single then
        S[d] = FPConvert(D[m], FPSCR[]);
    else
        D[d] = FPConvert(S[m], FPSCR[]);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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)
(Armv8.2)

```
VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(Armv8.2)

```
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>
```

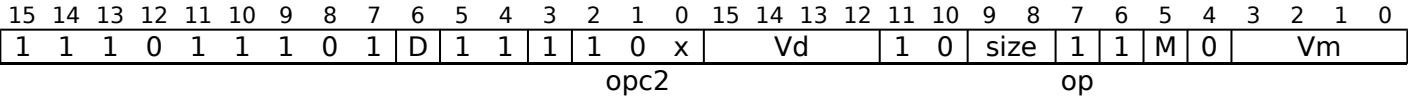
```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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) (Armv8.2)

VCVT{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar (opc2 == 101 && size == 01) (Armv8.2)

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 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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);
                [m]<15:0>, 0, unsigned, FPSCR, rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
                [m], 0, unsigned, FPSCR, rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
                [m], 0, unsigned, FPSCR, rounding);
        else
            case esize of
                when 16
                    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);[m], 0, unsigned, FPSCR,
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                    [m], 0, unsigned, FPSCR, rounding);
                when 64
                    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);[m], 0, unsigned, FPSCR, rounding);

```

Internal version only: isa [v01_19](#)~~v01_15~~, pseudocode [v2020-09_xml](#)~~v2020-06_rel~~, sve [v2020-09_rc3](#)~~v2020-06-29-gc9614a3~~; Build timestamp: [2020-09-30T21:3536](#)~~2020-07-03T11:3536~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 [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	0	0	0	Vd			1 0		size	op	1	M	0	Vm					
cond								opc2																							

Half-precision scalar (size == 01)

(Armv8.2)

`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 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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);

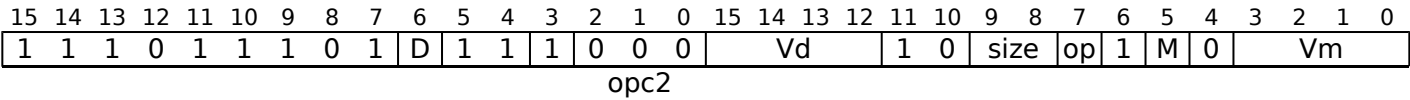
```

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)
(Armv8.2)

```
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 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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(); 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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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 [FPXCR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

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

A1

31	30	29	28	27	26	25	24	23	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)
(Armv8.2)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(Armv8.2)

```
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);
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)
(Armv8.2)

```
VCVT{<c>}{<q>}.F16.<dt> <Sdm>, <Sdm>, #<fbits>
```

Half-precision scalar (op == 1 && sf == 01)
(Armv8.2)

```
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);
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[],[d]<15:0>, frac_bits, unsigned);
                S[d] = Extend(result, 32, unsigned);
            when 32
                result = FPToFixed(S[d], frac_bits, unsigned, FPSCR[],[d], frac_bits, unsigned, FPSCR, FP);
                S[d] = Extend(result, 32, unsigned);
            when 64
                result = FPToFixed(D[d], frac_bits, unsigned, FPSCR[],[d], frac_bits, unsigned, FPSCR, FP);
                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[],[d]<size-1:0>, frac_bits, unsigned);
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[d]<size-1:0>, frac_bits, unsigned, FPSCR[],[d]<size-1:0>, frac_bits, unsigned);
                when 64
                    D[d] = FixedToFP(D[d]<size-1:0>, frac_bits, unsigned, FPSCR[],[d]<size-1:0>, frac_bits, unsigned);

```

Internal version only: isa v01_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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) (Armv8.2)

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);
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) (Armv8.2)

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 = FPDcodeRM(RM); unsigned = (op == '0');
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);

```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt>	Is the data type for the elements of the destination, encoded in "op":
------	--

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    [m]<15:0>, 0, unsigned, FPSCR, rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    [m], 0, unsigned, FPSCR, rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding); [m], 0, unsigned, FPSCR, rounding);
```

Internal version only: isa **v01_19v01_15**, pseudocode **v2020-09_xmlv2020-06_rel**, sve **v2020-09_rc3v2020-06-29-gc9614a3**; Build timestamp: **2020-09-30T21:2020-07-03T11:3536**

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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);
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);
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(); CheckVFPEEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        (hp, FPSCR);
    else
        S[d] = FPConvert(hp, FPSCR[]);
        (hp, FPSCR);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        [m], FPSCR);
    else
        hp = FPConvert(S[m], FPSCR[]); [m], FPSCR);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa [v01_19v01_15](#), pseudocode [v2020-09_xmlv2020-06_rel](#), sve [v2020-09_rc3v2020-06-29-gc9614a3](#); Build timestamp: [2020-09-30T21:2020-07-03T11:3536](#)

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 (Armv8.6)

31	30	29	28	27	26	25	24	23	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 (Armv8.6)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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[]); [m], FPSCR);
```

(old)

htmldiff from-

(new)

VCVTM (floating-point)

Convert floating-point to integer with Round towards -Infinity converts a value in a register from floating-point to a 32-bit integer using the Round towards -Infinity rounding mode, and places the result in a second register.

Depending on settings in the [CPACR](#), [NSACR](#), [HCPTR](#), and [FPEXC](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	1	1	1	Vd				1	0	!= 00	op	1	M	0	Vm				
RM																size															

Half-precision scalar (size == 01) (Armv8.2)

```
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);
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) (Armv8.2)

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 = FPDecoderRM(RM); unsigned = (op == '0');
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);

```

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);
    [m]<15:0>, 0, unsigned, FPSCR, rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    [m], 0, unsigned, FPSCR, rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding); [m], 0, unsigned, FPSCR, rounding);
```

Internal version only: isa **v01_19v01_15**, pseudocode **v2020-09_xmlv2020-06_rel**, sve **v2020-09_rc3v2020-06-29-gc9614a3**; Build timestamp: **2020-09-30T21:20:07Z**; Build date: **2020-07-03T11:35:36Z**

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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) (Armv8.2)

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);
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)
(Armv8.2)

```
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);
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(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    [m]<15:0>, 0, unsigned, FPSCR, rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    [m], 0, unsigned, FPSCR, rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);[m], 0, unsigned, FPSCR, rounding);
```

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 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) (Armv8.2)

```
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);
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) (Armv8.2)

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 = FPDecoderRM(RM); unsigned = (op == '0');
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);

```

CONSTRAINED UNPREDICTABLE behavior

If `InITBlock()`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<q> See *Standard assembler syntax fields*.

<dt> Is the data type for the elements of the destination, encoded in “op”:

op	<dt>
0	U32
1	S32

<Sd> Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.

<Sm> Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.

<Dm> Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = FPToFixed(S[m]<15:0>, 0, unsigned, FPSCR[], rounding);
    [m]<15:0>, 0, unsigned, FPSCR, rounding);
  when 32
    S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
    [m], 0, unsigned, FPSCR, rounding);
  when 64
    S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding); [m], 0, unsigned, FPSCR, rounding);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:07Z; Build date: 2020-07-03T11:35:36Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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)
(Armv8.2)

```
VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>
```

Half-precision scalar (opc2 == 101 && size == 01)
(Armv8.2)

```
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>
```

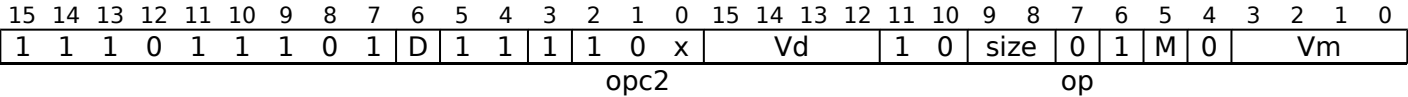
```
if opc2 != '000' && opc2 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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) (Armv8.2)

VCVTR{<c>}{<q>}.U32.F16 <Sd>, <Sm>

Half-precision scalar (opc2 == 101 && size == 01) (Armv8.2)

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 != '10x' then SEE "Related encodings";
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
to_integer = (opc2<2> == '1');
if to_integer then
    unsigned = (opc2<0> == '0');
    rounding = if op == '1' then FPRounding_ZERO else FPRoundingMode(FPSCR[]);
(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[]);
(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);
                [m]<15:0>, 0, unsigned, FPSCR, rounding);
            when 32
                S[d] = FPToFixed(S[m], 0, unsigned, FPSCR[], rounding);
                [m], 0, unsigned, FPSCR, rounding);
            when 64
                S[d] = FPToFixed(D[m], 0, unsigned, FPSCR[], rounding);
                [m], 0, unsigned, FPSCR, rounding);
        else
            case esize of
                when 16
                    bits(16) fp16 = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);[m], 0, unsigned, FPSCR,
                    S[d] = Zeros(16):fp16;
                when 32
                    S[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);
                    [m], 0, unsigned, FPSCR, rounding);
                when 64
                    D[d] = FixedToFP(S[m], 0, unsigned, FPSCR[], rounding);[m], 0, unsigned, FPSCR, rounding);

```

Internal version only: isa [v01_19](#)~~v01_15~~, pseudocode [v2020-09_xml](#)~~v2020-06_rel~~, sve [v2020-09_rc3](#)~~v2020-06-29-gc9614a3~~; Build timestamp: [2020-09-30T21:3536](#)~~2020-07-03T11:3536~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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);
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);
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(); CheckVFPEEnabled(TRUE);
    bits(16) hp;
    if convert_from_half then
        hp = S[m]<lowbit+15:lowbit>;
        if uses_double then
            D[d] = FPConvert(hp, FPSCR[]);
        (hp, FPSCR);
    else
        S[d] = FPConvert(hp, FPSCR[]);
        (hp, FPSCR);
    else
        if uses_double then
            hp = FPConvert(D[m], FPSCR[]);
        [m], FPSCR);
    else
        hp = FPConvert(S[m], FPSCR[]); [m], FPSCR);
        S[d]<lowbit+15:lowbit> = hp;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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 (Armv8.6)

31	30	29	28	27	26	25	24	23	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 (Armv8.6)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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();
    CheckVFPEEnabled(TRUE);

    S[d]<31:16> = FPConvertBF(S[m], FPSCR[]);[m], FPSCR);
```

(old)

htmldiff from-

(new)

VDIV

VDIV 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) (Armv8.2)

```
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;
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) (Armv8.2)

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;
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[]);
```

Internal version only: isa [v01_19v01-15](#), pseudocode [v2020-09_xmlv2020-06-rel](#), sve [v2020-09_rc3v2020-06-29-gc9614a3](#); Build timestamp: [2020-09-30T21:35:36](#)

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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');
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) (Armv8.2)

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; opl_neg = (op == '1');
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	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>

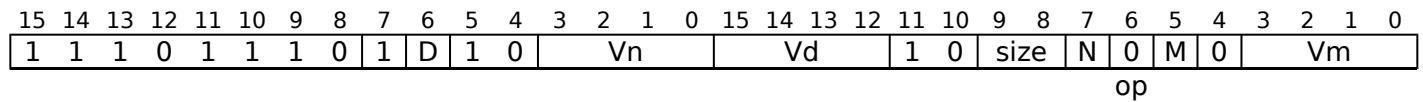
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; opl_neg = (op == '1');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
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) (Armv8.2)

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; opl_neg = (op == '1');
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": <table border="1" data-bbox="240 1444 435 1541"> <tr> <th>sz</th><th><dt></th></tr> <tr> <td>0</td><td>F32</td></tr> <tr> <td>1</td><td>F16</td></tr> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<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[]);
                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[]);
                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[]); D[m], FPSCR);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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');
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) (Armv8.2)

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; opl_neg = (op == '1');
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	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>

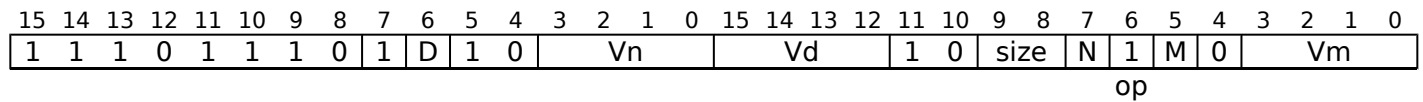
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; opl_neg = (op == '1');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
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) (Armv8.2)

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');
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": <table border="1" style="margin: 10px auto; border-collapse: collapse; text-align: center;"> <tr> <th style="padding: 2px 10px;">sz</th><th style="padding: 2px 10px;"><dt></th></tr> <tr> <td style="padding: 2px 10px;">0</td><td style="padding: 2px 10px;">F32</td></tr> <tr> <td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">F16</td></tr> </table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<Qd>	Is the 128-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field as <Qd>*2.						
<Qn>	Is the 128-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field as <Qn>*2.						
<Qm>	Is the 128-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field as <Qm>*2.						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:Vn" field.						
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:Vm" field.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.						
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.						

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckAdvSIMDOrVFPEEnabled(TRUE, advsimd);
    if advsimd then // Advanced SIMD instruction
        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[]);
                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[]);
                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[]); D[m], FPSCR);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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) (Armv8.2)

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');
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) (Armv8.2)

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');
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:D); n = UInt(N:N); m = UInt(M:M);
```

CONSTRAINED UNPREDICTABLE behavior

If size == '01' && InITBlock(), then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as if it passes the Condition code check.
- The instruction executes as NOP. This means it behaves as if it fails the Condition code check.

Assembler Symbols

<c>	See Standard assembler syntax fields .
<q>	See Standard assembler syntax fields .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sn>	Is the 32-bit name of the first SIMD&FP source register, encoded in the "Vn:N" field.
<Sm>	Is the 32-bit name of the second SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:D" field.
<Dn>	Is the 64-bit name of the first SIMD&FP source register, encoded in the "N:N" field.
<Dm>	Is the 64-bit name of the second SIMD&FP source register, encoded in the "M:M" 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[]);
```

(old)

htmldiff from-

(new)

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 [FPXCR](#) registers, and the Security state and PE mode in which the instruction is executed, an attempt to execute the instruction might be UNDEFINED, or trapped to Hyp mode. For more information see [Enabling Advanced SIMD and floating-point support](#).

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

A1

31	30	29	28	27	26	25	24	23	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)
(Armv8.2)

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');
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) (Armv8.2)

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');
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(); CheckVFPEEnabled(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[]);
```

(old)

htmldiff from-

(new)

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

(Armv8.3)

31	30	29	28	27	26	25	24	23	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

(Armv8.3)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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);
(fltval, FPSCR, FALSE);
FPSCR<31:28> = '0':Z:'00';
S[d] = intval;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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(aligned);
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.

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 || 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.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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(()) then AccType_NORMAL then MemU[address+4,4] else MemU[address,4];
                data<63:32> = if BigEndian(AccType_NORMAL) then(()) 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;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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() then word1:word2 else word2:word1;
            if wback then AccType_ATOMIC) then word1:word2 else word2:word1;
            if wback then R[n] = if add then R[n]+imm32 else R[n]-imm32;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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) (Armv8.2)

```
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);
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)
(Armv8.2)

```
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);
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 <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
.64	Is an optional data size specifier for 64-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.						
.32	Is an optional data size specifier for 32-bit memory accesses that can be used in the assembler source code, but is otherwise ignored.						
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.						
<Rn>	Is the general-purpose base register, encoded in the "Rn" field.						
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4. For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  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(()) then word1:word2 else word2:word1; AccType_ATOMIC) then word1:word2 else word2:word1
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:20:07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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) (Armv8.2)

```
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);
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) (Armv8.2)

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);
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>	<p>The label of the literal data item to be loaded.</p> <p>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.</p> <p>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.</p> <p>If the offset is zero or positive, imm32 is equal to the offset and add == TRUE.</p> <p>If the offset is negative, imm32 is equal to minus the offset and add == FALSE.</p>
+/-	Specifies the offset is added to or subtracted from the base register, defaulting to + if omitted and encoded in "U":

U	+/-
0	-
1	+

<imm> For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4.

For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.

The alternative syntax permits the addition or subtraction of the offset and the immediate offset to be specified separately, including permitting a subtraction of 0 that cannot be specified using the normal syntax. For more information, see *Use of labels in UAL instruction syntax*.

Operation

```

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    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(()) then word1:word2 else word2:word1; AccType_ATOMIC then word1:word2 else

```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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;
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	size		!= 00	N	0	M	0	Vm		
																						size		op							

Half-precision scalar (size == 01) (Armv8.2)

```
VMAXNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Single-precision scalar (size == 10)

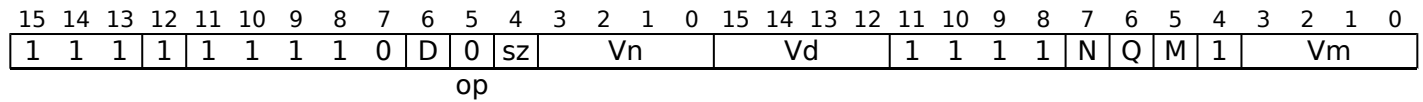
```
VMAXNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

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');
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;
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;

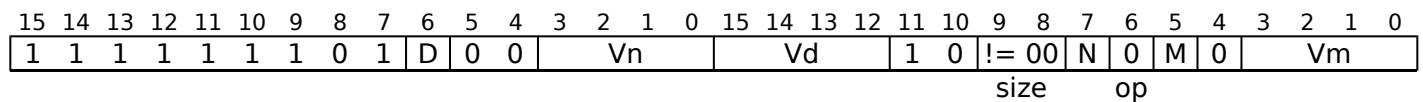
```

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) (Armv8.2)

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');
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(); CheckAdvSIMDorVFPEntabled(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] = FPMaNum(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) : FPMaNum(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] = FPMaNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMInNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMInNum(D[n], D[m], FPSCR[]);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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;
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) (Armv8.2)

```
VMINNM{<q>}.F16 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

Single-precision scalar (size == 10)

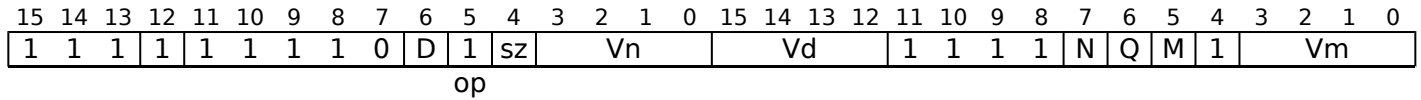
```
VMINNM{<q>}.F32 <Sd>, <Sn>, <Sm> // (Cannot be conditional)
```

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');
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;
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;

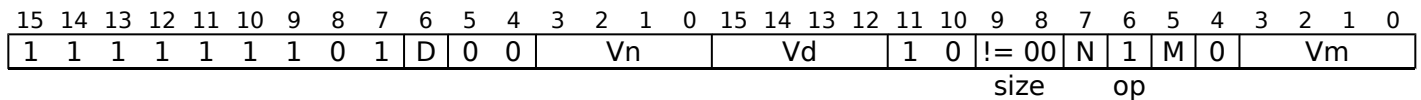
```

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) (Armv8.2)

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');
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(); CheckAdvSIMDorVFPEntabled(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] = FPMaNum(op1, op2, StandardFPSCRValue());
            else
                Elem[D[d+r], e, esize] = FPMiNum(op1, op2, StandardFPSCRValue());
else
    // VFP instruction
    case esize of
        when 16
            if maximum then
                S[d] = Zeros(16) : FPMaNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            else
                S[d] = Zeros(16) : FPMiNum(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
        when 32
            if maximum then
                S[d] = FPMaNum(S[n], S[m], FPSCR[]);
            else
                S[d] = FPMiNum(S[n], S[m], FPSCR[]);
        when 64
            if maximum then
                D[d] = FPMaNum(D[n], D[m], FPSCR[]);
            else
                D[d] = FPMiNum(D[n], D[m], FPSCR[]);
```


Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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');
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) (Armv8.2)

```
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>
```

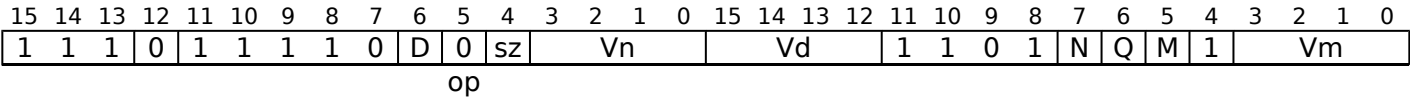
```
if size == '00' || (size == '01' && !HaveFP16Ext()) then UNDEFINED;
if size == '01' && cond != '1110' then UNPREDICTABLE;
if FPSCR.Len != '000' || FPSCR.Stride != '00' then UNDEFINED;
advsimd = FALSE; add = (op == '0');
case size of
  when '01' esize = 16; d = UInt(Vd:D); n = UInt(Vn:N); m = UInt(Vm:M);
  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>
```

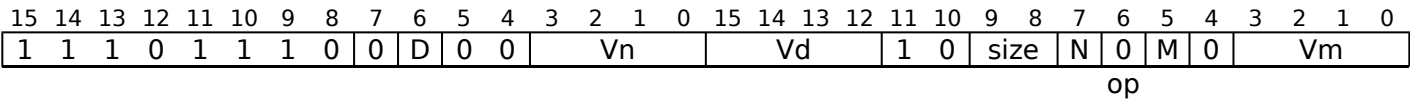
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = 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)
(Armv8.2)

```
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');
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 <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding A2, T1 and T2: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in "sz": <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<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] = FPAdd(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 [m]<15:0>, FPSCR) else
                S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
                [d]<15:0>, addend16, FPSCR);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else [m], FPSCR) else FPNeg(FPMul(S[n],
                S[d] = FPAdd(S[d], addend32, FPSCR[]);
                [d], addend32, FPSCR);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else [m], FPSCR) else FPNeg(FPMul(D[n],
                D[d] = FPAdd(D[d], addend64, FPSCR[]); [d], addend64, FPSCR);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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.

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');
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) (Armv8.2)

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');
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	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>

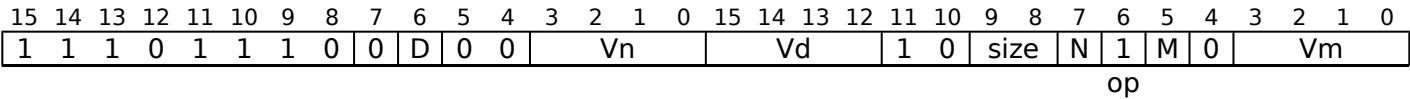
```
if Q == '1' && (Vd<0> == '1' || Vn<0> == '1' || Vm<0> == '1') then UNDEFINED;
if sz == '1' && !HaveFP16Ext() then UNDEFINED;
if sz == '1' && InITBlock() then UNPREDICTABLE;
advsimd = TRUE; add = (op == '0');
case sz of
  when '0' esize = 32; elements = 2;
  when '1' esize = 16; elements = 4;
d = 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)
(Armv8.2)

```
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');
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] = FPAdd(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 [m]<15:0>, FPSCR) else
                    S[d] = Zeros(16) : FPAdd(S[d]<15:0>, addend16, FPSCR[]);
                [d]<15:0>, addend16, FPSCR);
            when 32
                addend32 = if add then FPMul(S[n], S[m], FPSCR[]) else [m], FPSCR) else FPNeg(FPMul(S[n],
                    S[d] = FPAdd(S[d], addend32, FPSCR[]);
                [d], addend32, FPSCR);
            when 64
                addend64 = if add then FPMul(D[n], D[m], FPSCR[]) else [m], FPSCR) else FPNeg(FPMul(D[n],
                    D[d] = FPAdd(D[d], addend64, FPSCR[]); [d], addend64, FPSCR);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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;
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) (Armv8.2)

```
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;
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	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;
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)
(Armv8.2)

```
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;

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 <i>Standard assembler syntax fields</i> . This encoding must be unconditional. For encoding A2, T1 and T2: see <i>Standard assembler syntax fields</i> .						
<q>	See <i>Standard assembler syntax fields</i> .						
<dt>	Is the data type for the elements of the vectors, encoded in “sz”: <table><tr><th>sz</th><th><dt></th></tr><tr><td>0</td><td>F32</td></tr><tr><td>1</td><td>F16</td></tr></table>	sz	<dt>	0	F32	1	F16
sz	<dt>						
0	F32						
1	F16						
<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_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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.

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)
(ArmV8.2)

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;
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	1	M	0	Vm				
																op															

Half-precision scalar (size == 01) (Armv8.2)

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;
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

```

enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            [m]<15:0>, FPSCR);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                (product16), FPSCR);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                [d]<15:0>, product16, FPSCR);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                [m], FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    (product32), FPSCR);
                    when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                    [d]), product32, FPSCR);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                [m], FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    (product64), FPSCR);
                    when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                    [d]), product64, FPSCR);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);

```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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)
(Armv8.2)

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;
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) (Armv8.2)

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;
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

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            [m]<15:0>, FPSCR);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                (product16), FPSCR);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                [d]<15:0>, product16, FPSCR);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                [m], FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    (product32), FPSCR);
                    when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                    [d]), product32, FPSCR);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                [m], FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    (product64), FPSCR);
                    when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                    [d]), product64, FPSCR);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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) (Armv8.2)

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;
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) (Armv8.2)

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;
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

```
enumeration VFPNegMul {VFPNegMul_VNMLA, VFPNegMul_VNMLS, VFPNegMul_VNMUL};

if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    case esize of
        when 16
            product16 = FPMul(S[n]<15:0>, S[m]<15:0>, FPSCR[]);
            {m}<15:0>, FPSCR);
            case vtype of
                when VFPNegMul_VNMLA S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), FPNeg(product16), FPSCR[]);
                {product16}, FPSCR);
                when VFPNegMul_VNMLS S[d] = Zeros(16) : FPAdd(FPNeg(S[d]<15:0>), product16, FPSCR[]);
                {d}<15:0>, product16, FPSCR);
                when VFPNegMul_VNMUL S[d] = Zeros(16) : FPNeg(product16);
            when 32
                product32 = FPMul(S[n], S[m], FPSCR[]);
                {m}, FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA S[d] = FPAdd(FPNeg(S[d]), FPNeg(product32), FPSCR[]);
                    {product32}, FPSCR);
                    when VFPNegMul_VNMLS S[d] = FPAdd(FPNeg(S[d]), product32, FPSCR[]);
                    {d}, product32, FPSCR);
                    when VFPNegMul_VNMUL S[d] = FPNeg(product32);
            when 64
                product64 = FPMul(D[n], D[m], FPSCR[]);
                {m}, FPSCR);
                case vtype of
                    when VFPNegMul_VNMLA D[d] = FPAdd(FPNeg(D[d]), FPNeg(product64), FPSCR[]);
                    {product64}, FPSCR);
                    when VFPNegMul_VNMLS D[d] = FPAdd(FPNeg(D[d]), product64, FPSCR[]);
                    {d}, product64, FPSCR);
                    when VFPNegMul_VNMUL D[d] = FPNeg(product64);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:35Z2020-07-03T11:35:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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' && (!if (size == '01' && !HaveFP16Ext() || F == '0')) || size IN {'00', '11'}) then UNDEF
()) || size IN {'00', '11'}) then UNDEFINED;
floating_point = (F == '1');
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' && (!if (size == '01' && !HaveFP16Ext() || F == '0')) || size IN {'00', '11'}) then UNDEF
()) || size IN {'00', '11'}) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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) (Armv8.2)

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;
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)
(Armv8.2)

```
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;
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(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    [m]<15:0>, FPSCR, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    [m], FPSCR, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [m], FPSCR, rounding, exact);
```

Internal version only: isa ~~v01_19v01_15~~, pseudocode ~~v2020-09_xmlv2020-06_rel~~, sve ~~v2020-09_rc3v2020-06-29-gc9614a3~~; Build timestamp: ~~2020-09-30T21:2020-07-03T11:3536~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTM (floating-point)

Round floating-point to integer towards -Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards -Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	1	1	Vd				1	0	!= 00		0	1	M	0	Vm			
RM																size															

Half-precision scalar (size == 01) (Armv8.2)

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;
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	1	1	Vd				1	0	!= 00		0	1	M	0	Vm			
RM																size															

Half-precision scalar (size == 01)
(Armv8.2)

```
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;
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(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    [m]<15:0>, FPSCR, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    [m], FPSCR, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [m], FPSCR, rounding, exact);
```

VRINTN (floating-point)

Round floating-point to integer to Nearest rounds a floating-point value to an integral floating-point value of the same size using the Round to Nearest rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	1	D	1	1	1	0	0	1	Vd				1	0	!= 00	0	1	M	0	Vm				
RM																size															

Half-precision scalar (size == 01) (Armv8.2)

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;
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	1	Vd				1	0	!= 00	0	1	M	0	Vm				
RM																size															

Half-precision scalar (size == 01)
(Armv8.2)

```
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;
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(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    [m]<15:0>, FPSCR, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    [m], FPSCR, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [m], FPSCR, rounding, exact);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

VRINTP (floating-point)

Round floating-point to integer towards +Infinity rounds a floating-point value to an integral floating-point value of the same size using the Round towards +Infinity rounding mode. A zero input gives a zero result with the same sign, an infinite input gives an infinite result with the same sign, and a NaN is propagated as for normal arithmetic.

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

A1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1 1 1 1 1 1 0 1									D	1 1 1 0				1 0		Vd				1 0		!= 00		0	1	M	0	Vm			
RM																size															

Half-precision scalar (size == 01) (Armv8.2)

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;
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	1	0	Vd				1	0	!= 00		0	1	M	0	Vm			
RM																size															

Half-precision scalar (size == 01)
(Armv8.2)

```
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;
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(); CheckVFPEEnabled(TRUE);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
    [m]<15:0>, FPSCR, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
    [m], FPSCR, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [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)
(Armv8.2)

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[]);
(FPSCR);
exact = FALSE;
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) (Armv8.2)

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[]);
(FPSCR);
exact = FALSE;
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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
      [m]<15:0>, FPSCR, rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
      [m], FPSCR, rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [m], FPSCR, rounding, exact);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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)
(Armv8.2)

```
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;
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) (Armv8.2)

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;
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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
    rounding = FPRoundingMode(FPSCR[]);
(FPSCR);
case esize of
  when 16
    S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
[m]<15:0>, FPSCR, rounding, exact);
  when 32
    S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
[m], FPSCR, rounding, exact);
  when 64
    D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact);[m], FPSCR, rounding, exact);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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) (Armv8.2)

```
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[]);
{FPSCR};
exact = FALSE;
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) (Armv8.2)

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[]);
(FPSCR);
exact = FALSE;
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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16
      S[d] = Zeros(16) : FPRoundInt(S[m]<15:0>, FPSCR[], rounding, exact);
      [m]<15:0>, FPSCR, rounding, exact);
    when 32
      S[d] = FPRoundInt(S[m], FPSCR[], rounding, exact);
      [m], FPSCR, rounding, exact);
    when 64
      D[d] = FPRoundInt(D[m], FPSCR[], rounding, exact); [m], FPSCR, rounding, exact);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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' && (!if (size == '01' && !HaveFP16Ext() || F == '0')) || size IN {'00', '11'}) then UNDEF
()) || size IN {'00', '11'}) then UNDEFINED;
floating_point = (F == '1');
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' && (!if (size == '01' && !HaveFP16Ext() || F == '0')) || size IN {'00', '11'}) then UNDEF
()) || size IN {'00', '11'}) then UNDEFINED;
if size == '01' && InITBlock() then UNPREDICTABLE;
floating_point = (F == '1');
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_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:20:07Z

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)

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)
(Armv8.2)

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;
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)
(Armv8.2)

```
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;
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 <i>Standard assembler syntax fields</i> .
<q>	See <i>Standard assembler syntax fields</i> .
<Sd>	Is the 32-bit name of the SIMD&FP destination register, encoded in the "Vd:D" field.
<Sm>	Is the 32-bit name of the SIMD&FP source register, encoded in the "Vm:M" field.
<Dd>	Is the 64-bit name of the SIMD&FP destination register, encoded in the "D:Vd" field.
<Dm>	Is the 64-bit name of the SIMD&FP source register, encoded in the "M:Vm" field.

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  case esize of
    when 16 S[d] = Zeros(16) : FPSqrt(S[m]<15:0>, FPSCR[]);
    [m]<15:0>, FPSCR);
    when 32 S[d] = FPSqrt(S[m], FPSCR[]);
    [m], FPSCR);
    when 64 D[d] = FPSqrt(D[m], FPSCR[]); [m], FPSCR);
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 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 || 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.

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 || 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.

T2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	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								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 $d + \text{regs} > 32$, 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.

T4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	0	0	Rn				Vd				0	0	1	0	size	align	Rm					

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.

For more information about the CONSTRAINED UNPREDICTABLE behavior of this instruction, see [Architectural Constraints on UNPREDICTABLE behaviors](#), and particularly [VST1 \(multiple single elements\)](#).

Related encodings: See [Advanced SIMD element or structure load/store](#) for the T32 instruction set, or [Advanced SIMD element or structure load/store](#) for the A32 instruction set.

Assembler Symbols

<c> For encoding A1, A2, A3 and A4: see [Standard assembler syntax fields](#). This encoding must be unconditional.

For encoding T1, T2, T3 and T4: see [Standard assembler syntax fields](#).

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

<size> Is the data size, encoded in "size":

size	<size>
00	8
01	16
10	32
11	64

<list> Is a list containing the 64-bit names of the SIMD&FP registers.
The list must be one of:

{ <Dd> }

Single register. Selects the A1 and T1 encodings of the instruction.

{ <Dd>, <Dd+1> }

Two single-spaced registers. Selects the A2 and T2 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2> }

Three single-spaced registers. Selects the A3 and T3 encodings of the instruction.

{ <Dd>, <Dd+1>, <Dd+2>, <Dd+3> }

Four single-spaced registers. Selects the A4 and T4 encodings of the instruction.

The register <Dd> is encoded in the "D:Vd" field.

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

<align> Is the optional alignment.

Whenever <align> is omitted, the standard alignment is used, see [Unaligned data access](#), and is encoded in the "align" field as 0b00.

Whenever <align> is present, the permitted values are:

64

64-bit alignment, encoded in the "align" field as 0b01.

128

128-bit alignment, encoded in the "align" field as 0b10. Available only if <list> contains two or four registers.

256

256-bit alignment, encoded in the "align" field as 0b11. Available only if <list> contains four registers.

: is the preferred separator before the <align> value, but the alignment can be specified as @<align>, see [Advanced SIMD addressing mode](#).

<Rm> Is the general-purpose index register containing an offset applied after the access, encoded in the "Rm" field.

For more information about <Rn>, !, and <Rm>, see [Advanced SIMD addressing mode](#).

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckAdvSIMDEnabled();
  address = R[n]; 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(()) then data<63:32> else data<31:0>; AccType_NORMAL) then c
        MemU[address+4,4] = if BigEndian(AccType_NORMAL) then data<31:0> else data<63:32>;
        () 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;
```

Internal version only: isa v01_19v01_15, pseudocode v2020-09_xmlv2020-06_rel, sve v2020-09_rc3v2020-06-29-gc9614a3; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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	P == '1' && U == '0' && 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
            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(()) then AccType_ATOMIC then D[d+r]<63:32> else D[d+r]<31:0>;
            MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then(()) 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;
```

Internal version only: isa ~~v01_19~~~~v01_15~~, pseudocode ~~v2020-09_xml~~~~v2020-06_rel~~, sve ~~v2020-09_rc3~~~~v2020-06-29-gc9614a3~~; Build timestamp: ~~2020-09-30T21:35:36~~~~2020-07-03T11:35:36~~

Copyright © 2010-2020 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)
(Armv8.2)

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);
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)
(Armv8.2)

```
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);
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": <table><tr><th>U</th><th>+/-</th></tr><tr><td>0</td><td>-</td></tr><tr><td>1</td><td>+</td></tr></table>	U	+/-	0	-	1	+
U	+/-						
0	-						
1	+						
<imm>	For the single-precision scalar or double-precision scalar variants: is the optional unsigned immediate byte offset, a multiple of 4, in the range 0 to 1020, defaulting to 0, and encoded in the "imm8" field as <imm>/4. For the half-precision scalar variant: is the optional unsigned immediate byte offset, a multiple of 2, in the range 0 to 510, defaulting to 0, and encoded in the "imm8" field as <imm>/2.						

Operation

```
if ConditionPassed() then
  EncodingSpecificOperations(); CheckVFPEEnabled(TRUE);
  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(()) then AccType_ATOMIC then D[d]<63:32> else D[d]<31:0>;
      MemA[address+4,4] = if BigEndian(AccType_ATOMIC) then() then D[d]<31:0> else D[d]<63:32>;
```

Internal version only: isa v01_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-gc9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

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;
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) (Armv8.2)

```
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;
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;
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)
(Armv8.2)

```
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;
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_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-ge9614a3; Build timestamp: 2020-09-30T21:20:07+03:36

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)	htmldiff from-	(new)
-------	----------------	-------

(old)

htmldiff from-

(new)

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	op1	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

P:W	Decode fields		Instruction Details	
	o1	Rn	op2	
10	0		01	STRH (immediate) — offset
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	
000			AND, ANDS (register)
001			EOR, EORS (register)
010	0	!= 1101	SUB, SUBS (register) — SUB
010	0	1101	SUB, SUBS (SP minus register) — SUB
010	1	!= 1101	SUB, SUBS (register) — SUBS
010	1	1101	SUB, SUBS (SP minus register) — SUBS
011			RSB, RSBS (register)
100	0	!= 1101	ADD, ADDS (register) — ADD
100	0	1101	ADD, ADDS (SP plus register) — ADD
100	1	!= 1101	ADD, ADDS (register) — ADDS
100	1	1101	ADD, ADDS (SP plus register) — ADDS
101			ADC, ADCS (register)
110			SBC, SBCS (register)
111			RSC, RSCS (register)

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 opc	Instruction Details
00	TST (register)
01	TEQ (register)
10	CMP (register)
11	CMN (register)

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 opc	Instruction Details
00	ORR, ORRS (register)
01	MOV, MOVS (register)
10	BIC, BICS (register)
11	MVN, MVNS (register)

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 op0	Decode fields op1	Instruction details
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 opc	Instruction Details
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 opc	Instruction Details
00	TST (register-shifted register)
01	TEQ (register-shifted register)
10	CMP (register-shifted register)
11	CMN (register-shifted register)

Logical Arithmetic (three register, register shift)

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	Decode fields 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			Instruction Details
opc	S	Rn	
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
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																							
cond							imm4		Rd		imm12																				

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	Architecture Version
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	Armv8.2
00000	xxxx00010001	Reserved hint, behaves as NOP	-
00000	xxxx00010010	TSB CSYNC	Armv8.4
00000	xxxx00010011	Reserved hint, behaves as NOP	-
00000	xxxx00010100	CSDB	-
00000	xxxx00010101	Reserved hint, behaves as NOP	-
00000	xxxx00011xxx	Reserved hint, behaves as NOP	-
00000	xxxx0001111x	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																								
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 opc	Instruction Details
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

Decode fields				Instruction Details	
P:W	o2	o1	Rn		
!= 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

Decode fields				Instruction Details	
P	o2	W	o1		
0	0	0	0	STR (register) — post-indexed	

Decode fields				Instruction Details
P	o2	W	o1	
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
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

Decode fields			Instruction Details
op1	B	op2	
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
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

Decode fields			Instruction Details
op1	Ra	op2	
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
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				Ra				Rm				0 0 0 1				Rn			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Ra	
!= 1111	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				Rd				lsb				0 0 1			Rn					
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
Rn	
!= 1111	BFI

Decode fields Rn	Instruction Details
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	imm12												1	1	1	1	imm4			
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields cond	Instruction Details
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 U	Instruction Details
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 cond	op0	Instruction details
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		LDMDA, 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	0xxxxxxxxxxxxxxxxx	LDM (User registers)
1	1	0	0		STMIB, STMFA
1	1	0	1		LDMIB, LDMED
		1	1	1xxxxxxxxxxxxxxxxx	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		Instruction Details
cond	H	
!= 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	0x0		UNALLOCATED
	10	0x0		UNALLOCATED
	11			Supervisor call
1111	!=	1x1		Unconditional Advanced SIMD and floating-point instructions
!=	0x	1x1		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	1	0	Floating-point data-processing
1111				
!=	10	101	01	Floating-point data-processing Advanced SIMD and System register 32-bit move
1111				
!=	10	11	0	UNALLOCATED
1111				

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	VSELEQ, VSELGE, VSELGT, VSELVS
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	1	0	op1	D	op2				Vn				Vd			1	op3	0	op4	N	Q	M	U				Vm

Decode fields						Instruction Details		Architecture Version
op1	op2	op3	op4	Q	U			
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector		Armv8.3
x1	0x	0	0	0	1	UNALLOCATED		-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector		Armv8.3
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		Armv8.6
00	00	1	0	1	1	UNALLOCATED		-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector		Armv8.6
00	00	1	1	0	1	UNALLOCATED		-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector		Armv8.6
00	00	1	1	1	1	UNALLOCATED		-
00	01	1	0			UNALLOCATED		-
00	01	1	1			UNALLOCATED		-
00	10	0	0		1	VFMAL (vector)		Armv8.2
00	10	0	1			UNALLOCATED		-
00	10	1	0	0		UNALLOCATED		-
00	10	1	0	1	0	VSMMLA		Armv8.6
00	10	1	0	1	1	VUMMLA		Armv8.6
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector		Armv8.2
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector		Armv8.2
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector		Armv8.2
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector		Armv8.2
00	11	0	0		1	VFMAb, VFMAbT (BFloat16, vector)		Armv8.6
00	11	0	1			UNALLOCATED		-
00	11	1	0			UNALLOCATED		-
00	11	1	1			UNALLOCATED		-
01	10	0	0		1	VFMSL (vector)		Armv8.2
01	10	0	1			UNALLOCATED		-
01	10	1	0	0		UNALLOCATED		-
01	10	1	0	1	0	VUSMMLA		Armv8.6
01	10	1	0	1	1	UNALLOCATED		-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector		Armv8.6
01	10	1	1		1	UNALLOCATED		-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector		Armv8.6
01	11	0	1			UNALLOCATED		-

Decode fields						Instruction Details	Architecture Version
op1	op2	op3	op4	Q	U		
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	Armv8.3
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 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	Instruction Details
op	
0	VMAXNM
1	VMINNM

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	Instruction Details	Architecture Version
size op		
01	UNALLOCATED	-
10	0 VMOVX	Armv8.2
10	1 VINS	Armv8.2
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

Decode fields			Instruction Details	
o1	RM	size	op	
0		!= 00	1	UNALLOCATED
0	00		0	VRINTA (floating-point)
0	01		0	VRINTN (floating-point)
0	10		0	VRINTP (floating-point)
0	11		0	VRINTM (floating-point)
1	00			VCVTA (floating-point)
1	01			VCVTN (floating-point)
1	10			VCVTP (floating-point)
1	11			VCVTM (floating-point)

Decode fields			Instruction Details	
o1	RM			
0	00			VRINTA (floating-point)
0	01			VRINTN (floating-point)
0	10			VRINTP (floating-point)
0	11			VRINTM (floating-point)
1	00			VCVTA (floating-point)
1	01			VCVTN (floating-point)
1	10			VCVTP (floating-point)
1	11			VCVTM (floating-point)

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		Architecture Version	
op1	op2	Q	U				
0			0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point		Armv8.3	
0	00		1	VFMAL (by scalar)		Armv8.2	
0	01		1	VFMSL (by scalar)		Armv8.2	
0	10		1	UNALLOCATED		-	
0	11		1	VFMAB, VFMA (BFloat16, by scalar)		Armv8.6	
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point		Armv8.3	
1			1	UNALLOCATED		-	
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point		Armv8.3	

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		Architecture Version	
op1	op2	op4	Q	U				
0	00	0			UNALLOCATED		-	
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector		Armv8.6	

Decode fields					Instruction Details	Architecture Version
op1	op2	op4	Q	U		
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	Armv8.6
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	Armv8.2
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	Armv8.2
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	Armv8.2
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	Armv8.2
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	Armv8.6
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	Armv8.6
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	Armv8.6
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	Armv8.6
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															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					Rn	size	imm8	Instruction Details
P	U	W	L					
0	0	1						UNALLOCATED
0	1					0x		UNALLOCATED
0	1		0			10		VSTM, VSTMDB, VSTMIA
0	1		0			11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA
0	1		0			11	xxxxxxxx1	FSTMDBX, FSTMIAX — Increment After
0	1		1			10		VLDM, VLDMDB, VLDMIA
0	1		1			11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA
0	1		1			11	xxxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After
1		0	0					VSTR
1		0	1	!= 1111				VLDR (immediate)
1	0	1				0x		UNALLOCATED
1	0	1	0			10		VSTM, VSTMDB, VSTMIA
1	0	1	0			11	xxxxxxxx0	VSTM, VSTMDB, VSTMIA
1	0	1	0			11	xxxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before
1	0	1	1			10		VLDM, VLDMDB, VLDMIA
1	0	1	1			11	xxxxxxxx0	VLDM, VLDMDB, VLDMIA

Decode fields						Instruction Details	
P	U	W	L	Rn	size	imm8	
1	0	1	1		11	XXXXXXXX1	FLDM*X (FLDMDDBX, FLDMIAX) — Decrement Before
1		0	1	1111			VLDR (literal)
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																				
cond																															

The following constraints also apply to this encoding: cond != 1111 && P:U:D:W != 00x0 && cond != 1111

Decode fields						Instruction Details	
P:U:W	D	L	Rn	CRd	cp15		
!= 000	0			!= 0101	0	UNALLOCATED	
!= 000	0	1	1111	0101	0	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 Floating-point SIMD and System register 32-bit move 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											1		op1					1				
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
!= 1111					1110				op0											10				op1			0				

Decode fields		Instruction details		Architecture version
op0	op1			
0001x11	0001	Floating-point data-processing (two registers) UNALLOCATED		-
0001x11	0010	VMOV (between general-purpose register and half-precision) Floating-point move immediate		Armv8.2
000!= 1x11	010	VMOV (between general-purpose register and single-precision)		Floating-point data-processing (three registers)
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 movedata-processing special(two registerregisters)

These instructions are under [Advanced SIMD and System register 32-bit moveFloating-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	1	1	L		reg			Rt		1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)	(0)
cond																															
31	30	29	28	27	26	25	24	23	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										
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

Decode fields	Instruction Details
L	
0	VMSR
1	VMRS

o1	opc2	size	o3	Instruction Details	Architecture Version
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000		1	VABS	-
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	001		0	VNEG	-
0	001		1	VSQRT	-
0	010		0	VCVTB — half-precision to double-precision	-
0	010	01		UNALLOCATED	-
0	010		1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	Armv8.6
0	011	01	1	VCVTT (BFloat16)	Armv8.6
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		0	VCMP — A1	-
0	100		1	VCMPE — A1	-
0	101		0	VCMP — A2	-
0	101		1	VCMPE — A2	-
0	110		0	VRINTR	-
0	110		1	VRINTZ (floating-point)	-
0	111		0	VRINTX (floating-point)	-
0	111	01	1	UNALLOCATED	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000			VCVT (integer to floating-point, floating-point)	-

Decode fields				Instruction Details	Architecture Version
o1	opc2	size	o3		
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	Armv8.3
1	01x			VCVT (between floating-point and fixed-point, floating-point)	-
1	100		0	VCVTR	-
1	100		1	VCVT (floating-point to integer, floating-point)	-
1	101		0	VCVTR	-
1	101		1	VCVT (floating-point to integer, floating-point)	-
1	11x			VCVT (between floating-point and fixed-point, floating-point)	-

Advanced Floating-point SIMD move 8/16/32-bit element move/duplicate immediate

These instructions are under [Advanced SIMD and System register 32-bit move 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	opc1				L	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)
cond																																
31	30	29	28	27	26	25	24	23	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
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

Decode fields		Instruction Details	Architecture Version
size			
00		UNALLOCATED	-
01		VMOV (immediate) — half-precision scalar	Armv8.2
10		VMOV (immediate) — single-precision scalar	-
11		VMOV (immediate) — double-precision scalar	-

System Floating-point register data-processing 32-bit (three move registers)

These instructions are under [Advanced SIMD and System register 32-bit move 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	opc1				L	CRn				Rt				1	1	1	cp15	opc2		1	CRm			
cond																															
31	30	29	28	27	26	25	24	23	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
L	
0	MCR
1	MRC

Decode fields			Instruction Details
o0:o1	size	o2	
!= 111	00		UNALLOCATED
000		0	VMLA (floating-point)
000		1	VMLS (floating-point)
001		0	VNMLS
001		1	VNMLA
010		0	VMUL (floating-point)
010		1	VNMUL
011		0	VADD (floating-point)
011		1	VSUB (floating-point)
100		0	VDIV
101		0	VFNMS
101		1	VFNMA
110		0	VFMA
110		1	VFMS

Floating-point Advanced data-processing 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												101								op1								01							

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)

Decode fields	Instruction details	Architecture version
op0	op1	
000	000	UNALLOCATED
000	001	VMOV (between general-purpose register and half-precision)
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 data-processing move (two special registers) register

These instructions are under [Floating-point data-processing](#) [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	D	1	1	o1	opc2			Vd		1	0	size	o3	1	M	0										
cond																															
31	30	29	28	27	26	25	24	23	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)	(0)	(0)	(0)	(0)	(0)
cond																															

The following constraints also apply to this encoding: cond != 1111 && cond != 1111

o1	Decode fields opc2	size	o3	Instruction Details	Architecture Version
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000		1	VABS	-
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	001		0	VNEG	-
0	001		1	VSQRT	-
0	010		0	VCVTB — half-precision to double-precision	-
0	010	01		UNALLOCATED	-
0	010		1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	Armv8.6
0	011	01	1	VCVTT (BFloat16)	Armv8.6
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		0	VCMP — A1	-
0	100		1	VCMPE — A1	-
0	101		0	VCMP — A2	-
0	101		1	VCMPE — A2	-
0	110		0	VRINTR	-
0	110		1	VRINTZ (floating-point)	-
0	111		0	VRINTX (floating-point)	-
0	111	01	1	UNALLOCATED	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000			VCVT (integer to floating-point, floating-point)	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	Armv8.3
1	01x			VCVT (between floating-point and fixed-point, floating-point)	-
1	100		0	VCVTR	-
1	100		1	VCVT (floating-point to integer, floating-point)	-
1	101		0	VCVTR	-
1	101		1	VCVT (floating-point to integer, floating-point)	-
1	11x			VCVT (between floating-point and fixed-point, floating-point)	-

Decode fields	Instruction Details
L	
0	VMSR
1	VMRS

Floating-point Advanced SIMD 8/16/32-bit element move immediate/duplicate

These instructions are under [Floating-point data-processing 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	D	1	1			imm4H			Vd		1	0	size	(0)	0	(0)	0					imm4L				
cond																															
31	30	29	28	27	26	25	24	23	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	Architecture Version
size		
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	Armv8.2
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

Decode fields	Instruction Details
opc1 L opc2	
0xx 0	VMOV (general-purpose register to scalar)
0xx 1	VMOV (scalar to general-purpose register)
1xx 0 0x	VDUP (general-purpose register)
1xx 0 1x	UNALLOCATED

Floating-point System data-processing register (three 32-bit registers) move

These instructions are under [Floating-point data-processing 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	o0	opc1	D	o1	CRn	VnRt		Vd			1 01		cp15	N	opc2	o2	1	MCRm	0	Vm				
cond																															

The following constraints also apply to this encoding: cond != 1111 && o0:D;o1 != 1x11 && cond != 1111

Decode fields	Instruction Details
o0:o1 L size o2	
!= 1110 00	UNALLOCATED
000 1 0	VMLA (floating-point) MRC
000 1 1	VMLS (floating-point)
001 0 0	VNMLS
001 0 1	VNMLA
010 0 0	VMUL (floating-point)
010 0 1	VNMUL
011 0 0	VADD (floating-point)
011 0 1	VSUB (floating-point)

Decode fields			Instruction Details
o0:o1	size	o2	
100		0	VDIV
101		0	VFNMMS
101		1	VFNMMA
110		0	VFMA
110		1	VFMS

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
1	1	1	1	0							op1																				

Decode fields		Instruction details
op0	op1	
00x00		Miscellaneous
01x01		Advanced SIMD data-processing
1xx1x	1	Memory hints and barriers
10010	0	Advanced SIMD element or structure load/store
10111	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
1	1	1	1	0	0						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	Armv8.1
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					mode	Instruction Details
imod	M	op	I	F		
		1	0	0	0XXXX	SETEND
		0				CPS, CPSID, CPSIE
		1	0	0	1XXXX	UNALLOCATED
		1	0	1		UNALLOCATED
		1	1			UNALLOCATED

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	Architecture Version
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	-

U	Decode fields			o1	Instruction Details	Architecture Version
	size	opc	Q			
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) — 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	-

Decode fields				Instruction Details		Architecture Version
U	size	opc	Q	o1		
1	10	1100		0	SHA256SU1	-
1		1011		1	VQRDMLAH	Armv8.1
1	11	0001		1	VBIF	-
1		1100		1	VQRDMLSH	Armv8.1
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
1	1	1	1	0	0	1		op0	1			op1									op2				op3		0				

Decode fields				Instruction details	
op0	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](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	0	1	1	1	D	1	1	size	opc1	Vd			0	opc2			Q	M	0	Vm							

Decode fields				Instruction Details		Architecture Version
size	opc1	opc2	Q			
	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)		-

size	Decode fields		Q	Instruction Details	Architecture Version
	opc1	opc2			
	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)	Armv8.6
	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)	-
	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](#).

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	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

Decode fields opc	Instruction Details
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

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 VMLSL (integer)
1	0100 VRADDHN
1	0110 VRSUBHN
	11x0 VMULL (integer and polynomial)
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	Architecture Version
	000x VMLA (by scalar)	-
0	0011 VQDMLAL	-
	0010 VMLAL (by scalar)	-

Decode fields Q	opc	Instruction Details	Architecture Version
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	VQRDMULH	-
1	1011	UNALLOCATED	-
	1110	VQRDMLAH	Armv8.1
	1111	VQRDMLSH	Armv8.1

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
000xxxxxxxxxxx0	Advanced SIMD one register and modified immediate
!= 000xxxxxxxxxxx0	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 cmode	op	Instruction Details
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
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
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	
0	0	Reserved hint, behaves as NOP
0	1	PLI (register)
1	0	PLD, PLDW (register) — preload write
1	1	PLD, PLDW (register) — preload read

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	
0	000x	VST4 (multiple 4-element structures)
0	0010	VST1 (multiple single elements) — A4
0	0011	VST2 (multiple 2-element structures) — A2
0	010x	VST3 (multiple 3-element structures)
0	0110	VST1 (multiple single elements) — A3
0	0111	VST1 (multiple single elements) — A1
0	100x	VST2 (multiple 2-element structures) — A1
0	1010	VST1 (multiple single elements) — A2
1	000x	VLD4 (multiple 4-element structures)
1	0010	VLD1 (multiple single elements) — A4
1	0011	VLD2 (multiple 2-element structures) — A2
1	010x	VLD3 (multiple 3-element structures)
	1011	UNALLOCATED
1	0110	VLD1 (multiple single elements) — A3
1	0111	VLD1 (multiple single elements) — A1
	11xx	UNALLOCATED
1	100x	VLD2 (multiple 2-element structures) — A1
1	1010	VLD1 (multiple single elements) — A2

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					

Decode fields			Instruction Details
L	N	a	
0			UNALLOCATED
1	00		VLD1 (single element to all lanes)
1	01		VLD2 (single 2-element structure to all lanes)
1	10	0	VLD3 (single 3-element structure to all lanes)
1	10	1	UNALLOCATED
1	11		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](#).

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
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	
0	00	00	VST1 (single element from one lane) — A1
0	00	01	VST2 (single 2-element structure from one lane) — A1
0	00	10	VST3 (single 3-element structure from one lane) — A1
0	00	11	VST4 (single 4-element structure from one lane) — A1
0	01	00	VST1 (single element from one lane) — A2
0	01	01	VST2 (single 2-element structure from one lane) — A2
0	01	10	VST3 (single 3-element structure from one lane) — A2
0	01	11	VST4 (single 4-element structure from one lane) — A2
0	10	00	VST1 (single element from one lane) — A3
0	10	01	VST2 (single 2-element structure from one lane) — A3
0	10	10	VST3 (single 3-element structure from one lane) — A3
0	10	11	VST4 (single 4-element structure from one lane) — A3
1	00	00	VLD1 (single element to one lane) — A1
1	00	01	VLD2 (single 2-element structure to one lane) — A1
1	00	10	VLD3 (single 3-element structure to one lane) — A1
1	00	11	VLD4 (single 4-element structure to one lane) — A1
1	01	00	VLD1 (single element to one lane) — A2
1	01	01	VLD2 (single 2-element structure to one lane) — A2
1	01	10	VLD3 (single 3-element structure to one lane) — A2
1	01	11	VLD4 (single 4-element structure to one lane) — A2
1	10	00	VLD1 (single element to one lane) — A3
1	10	01	VLD2 (single 2-element structure to one lane) — A3
1	10	10	VLD3 (single 3-element structure to one lane) — A3
1	10	11	VLD4 (single 4-element structure to one lane) — A3

Internal version only: isa **v01_19**~~v01_15~~, pseudocode **v2020-09_xml**~~v2020-06_rel~~, sve **v2020-09_rc3**~~v2020-06-29_gc9614a3~~; Build timestamp: **2020-09-30T21:35:36**~~2020-07-03T11:35:36~~

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old) **htmldiff from-** (new)

(old)

htmldiff from-

(new)

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)

Decode fields	Instruction Details
op	
1001	RSB, RSBS (immediate)
1010	CMP (register)
1011	CMN (register)
1100	ORR, ORRS (register)
1101	MUL, MULS
1110	BIC, BICS (register)
1111	MVN, MVNS (register)

Special data instructions and branch and exchange

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	1	op0									

Decode fields	Instruction details
op0	
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	Instruction Details
L	
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

Decode fields			Instruction Details
op	D:Rd	Rs	
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	Instruction Details
SP	
0	ADR
1	ADD, ADDS (SP plus immediate)

Miscellaneous 16-bit instructions

These instructions are under [16-bit](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
10	11		op0	op1	op2	op3									

op0	Decode fields op1	op2	op3	Instruction details	Architecture version
0000				Adjust SP (immediate)	-
0010				Extend	-
0110	00	0		SETPAN	Armv8.1
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	Instruction Details
S	
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		Instruction Details
U	B	
0	0	SXTH
0	1	SXTB
1	0	UXTH
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		CPS, CPSID, CPSIE

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	0x0		UNALLOCATED
	10	0x0		UNALLOCATED
	11			Advanced SIMD data-processing
0	0x	1x1		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	1	0	Floating-point data-processing
0	10	101	01	Floating-point data-processing Advanced SIMD and System register 32-bit move
01	10!= 11	111	0	Additional Advanced SIMD and floating-point instructions 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							

Decode fields					Instruction Details	Architecture Version
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) — 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	-

U	Decode fields			Q	o1	Instruction Details	Architecture Version
	size	opc					
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) — 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	Armv8.1	
1	11	0001		1	VBIF	-	
1		1100		1	VQRDMLSH	Armv8.1	
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						

Decode fields				Instruction details	
op0	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									

Decode fields				Instruction Details	Architecture Version
size	opc1	opc2	Q		
	00	0000		VREV64	-

size	Decode fields		Q	Instruction Details	Architecture Version
	opc1	opc2			
	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		VC GT (immediate #0)	-
	01	x001		VC GE (immediate #0)	-
	01	x010		VC EQ (immediate #0)	-
	01	x011		VC LE (immediate #0)	-
	01	x100		VC LT (immediate #0)	-
	01	x110		VABS	-
	01	x111		VNEG	-
	01	0101	1	SHA1H	-
01	10	1100	1	VCVT (from single-precision to BFloat16, Advanced SIMD)	Armv8.6
	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)	-

Decode fields				Instruction Details	Architecture Version
size	opc1	opc2	Q		
	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](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	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	VMLSL (integer)
1	0100	VRADDHN
1	0110	VRSUBHN
	11x0	VMULL (integer and polynomial)
1	1001	UNALLOCATED

Decode fields	Instruction Details
U opc	
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](#).

15	14	13	12	11	10	9	8	7	6	5	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	Architecture Version
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	VQRDMULH
1	1011	UNALLOCATED
	1110	VQRDMLAH
	1111	VQRDMLSH
		Armv8.1
		Armv8.1

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		VRSHR
	!= 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

Decode fields						Instruction Details	
P	U	W	L	Rn	size	imm8	
0	0	1					UNALLOCATED
0	1				0x		UNALLOCATED
0	1		0		10		VSTM, VSTMDB, VSTMIA
0	1		0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA
0	1		0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Increment After
0	1		1		10		VLDM, VLDMDB, VLDMIA
0	1		1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA
0	1		1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Increment After
1		0	0				VSTR
1		0	1	!= 1111			VLDR (immediate)
1	0	1			0x		UNALLOCATED
1	0	1	0		10		VSTM, VSTMDB, VSTMIA
1	0	1	0		11	xxxxxxx0	VSTM, VSTMDB, VSTMIA
1	0	1	0		11	xxxxxxx1	FSTMDBX, FSTMIAX — Decrement Before
1	0	1	1		10		VLDM, VLDMDB, VLDMIA
1	0	1	1		11	xxxxxxx0	VLDM, VLDMDB, VLDMIA
1	0	1	1		11	xxxxxxx1	FLDM*X (FLDMDBX, FLDMIAX) — Decrement Before
1		0	1	1111			VLDR (literal)
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

Decode fields						Instruction Details	
P:U:W	D	L	Rn	CRd	cp15		
!= 000				!= 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 Floating-point SIMD and System register 32-bit movedata-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										1	op1										1			

15	14	13	12	11	10	9	8	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		Architecture version
op0	op1			
0001x11	0001	Floating-point data-processing (two registers) UNALLOCATED		-
0001x11	0010	VMOV (between general-purpose register and half-precision) Floating-point move immediate		Armv8.2
000! = 1x11	010	VMOV (between general-purpose register and single-precision)		Floating-point data-processing (three registers) -
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 [movedata-processing](#) [special\(two registerregisters\)](#)

These instructions are under [Advanced SIMD and System register 32-bit move](#)[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	1	1	L	reg				Rt				1	0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	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
L	
0	VMSR
1	VMRS

Decode fields				Instruction Details	Architecture Version
o1	opc2	size	o3		
		00		UNALLOCATED	-
0	000	01	0	UNALLOCATED	-
0	000		1	VABS	-
0	000	10	0	VMOV (register) — single-precision scalar	-
0	000	11	0	VMOV (register) — double-precision scalar	-
0	001		0	VNEG	-
0	001		1	VSQRT	-
0	010		0	VCVTB — half-precision to double-precision	-
0	010	01		UNALLOCATED	-
0	010		1	VCVTT — half-precision to double-precision	-
0	011	01	0	VCVTB (BFloat16)	Armv8.6
0	011	01	1	VCVTT (BFloat16)	Armv8.6
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		0	VCMP — T1	-
0	100		1	VCMP — T1	-
0	101		0	VCMP — T2	-

Decode fields				Instruction Details	Architecture Version
o1	opc2	size	o3		
0	101		1	VCMPE — T2	-
0	110		0	VRINTR	-
0	110		1	VRINTZ (floating-point)	-
0	111		0	VRINTX (floating-point)	-
0	111	01	1	UNALLOCATED	-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision	-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision	-
1	000			VCVT (integer to floating-point, floating-point)	-
1	001	01		UNALLOCATED	-
1	001	10		UNALLOCATED	-
1	001	11	0	UNALLOCATED	-
1	001	11	1	VJCVT	Armv8.3
1	01x			VCVT (between floating-point and fixed-point, floating-point)	-
1	100		0	VCVTR	-
1	100		1	VCVT (floating-point to integer, floating-point)	-
1	101		0	VCVTR	-
1	101		1	VCVT (floating-point to integer, floating-point)	-
1	11x			VCVT (between floating-point and fixed-point, floating-point)	-

Advanced Floating-point SIMD move 8/16/32-bit element move/duplicate immediate

These instructions are under [Advanced SIMD and System register 32-bit move 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	opc1			L	Vn				Rt				1	0	1	1	N	opc2		1	(0)	(0)	(0)	(0)
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	0	1	1	1	0	1	D	1	1	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			

Decode fields			Instruction Details
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

Decode fields		Instruction Details	Architecture Version
size			
00		UNALLOCATED	-
01		VMOV (immediate) — half-precision scalar	Armv8.2
10		VMOV (immediate) — single-precision scalar	-
11		VMOV (immediate) — double-precision scalar	-

System Floating-point register data-processing 32-bit (three move registers)

These instructions are under [Advanced SIMD and System register 32-bit move 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	opc1				L	CRn				Rt				1	1	1	cp15	opc2				1	CRm			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	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
I	
0	MCR
1	MRC

Decode fields	Instruction Details
o0:o1 size o2	
!= 111 00	UNALLOCATED
000	0 VMLA (floating-point)
000	1 VMLS (floating-point)
001	0 VNMLS
001	1 VNMLA
010	0 VMUL (floating-point)
010	1 VNMUL
011	0 VADD (floating-point)
011	1 VSUB (floating-point)
100	0 VDIV
101	0 VENMS
101	1 VENMA
110	0 VFMA
110	1 VFMS

Floating-pointAdvanced data-processingSIMD 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				101				op1				01			

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)

Decode fields	Instruction details	Architecture version
op0 op1		
000 000	UNALLOCATED	:
000 001	VMOV (between general-purpose register and half-precision)	Armv8.2
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 data-processingmove (two special registers)register

These instructions are under [Floating-point data-processingAdvanced 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	D	1	1	o1	opc2				Vd			1		0	size		o3	1	M	0	Vm			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	1	1	1	L	reg					Rt			1		0	1	0	(0)	(0)	(0)	1	(0)	(0)	(0)	(0)

Decode fields				Instruction Details				Architecture Version
o1	opc2	size	o3					
		00		UNALLOCATED				-
0	000	01	0	UNALLOCATED				-
0	000		1	VABS				-
0	000	10	0	VMOV (register) — single-precision scalar				-
0	000	11	0	VMOV (register) — double-precision scalar				-
0	001		0	VNEG				-
0	001		1	VSORT				-
0	010		0	VCVTB — half-precision to double-precision				-
0	010	01		UNALLOCATED				-
0	010		1	VCVTT — half-precision to double-precision				-
0	011	01	0	VCVTB (BFloat16)				Armv8.6
0	011	01	1	VCVTT (BFloat16)				Armv8.6
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		0	VCMP — T1				-
0	100		1	VCMPE — T1				-
0	101		0	VCMP — T2				-
0	101		1	VCMPE — T2				-
0	110		0	VRINTR				-
0	110		1	VRINTZ (floating-point)				-
0	111		0	VRINTX (floating-point)				-
0	111	01	1	UNALLOCATED				-
0	111	10	1	VCVT (between double-precision and single-precision) — single-precision to double-precision				-
0	111	11	1	VCVT (between double-precision and single-precision) — double-precision to single-precision				-
1	000			VCVT (integer to floating-point, floating-point)				-
1	001	01		UNALLOCATED				-
1	001	10		UNALLOCATED				-
1	001	11	0	UNALLOCATED				-
1	001	11	1	VJCVT				Armv8.3
1	01x			VCVT (between floating-point and fixed-point, floating-point)				-
1	100		0	VCVTR				-
1	100		1	VCVT (floating-point to integer, floating-point)				-
1	101		0	VCVTR				-
1	101		1	VCVT (floating-point to integer, floating-point)				-
1	11x			VCVT (between floating-point and fixed-point, floating-point)				-

Decode fields

Instruction Details

0	VMRS
1	VMRS

Floating-point Advanced SIMD 8/16/32-bit element move immediate/duplicate

These instructions are under [Floating-point data-processing](#)[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	D	1	1	imm4H				Vd				1	0	size		(0)	0	(0)	0	imm4L			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	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 size	Instruction Details	Architecture Version
00	UNALLOCATED	-
01	VMOV (immediate) — half-precision scalar	Armv8.2
10	VMOV (immediate) — single-precision scalar	-
11	VMOV (immediate) — double-precision scalar	-

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

Floating-pointSystem data-processingregister (three32-bit registers)move

These instructions are under [Floating-point data-processing](#)[Advanced SIMD and System register 32-bit move](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																
1	1	1	0	1	1	1	0	0	0	1	1	0	1	0	1							1																									
								OpC1DLolCRn								VnRt								Vd								101cp15Nopc2o21MCRm0								Vm							
																																size															

The following constraints also apply to this encoding: o0:D:o1 != 1x11

Decode fields			Instruction Details
o0:o1L	size	o2	
!= 1110	00		UNALLOCATED
0001		0	VMLA (floating-point)MRC
000		1	VMLS (floating-point)
001		0	VNMLS
001		1	VNMLA
010		0	VMUL (floating-point)
010		1	VNMUL
011		0	VADD (floating-point)
011		1	VSUB (floating-point)
100		0	VDIV
101		0	VFNMS
101		1	VFNMA
110		0	VFMA
110		1	VFMS

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> != 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	VSELEQ, VSELGE, VSELGT, VSELVS
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	Architecture Version
op1	op2	op3	op4	Q	U		
x1	0x	0	0	0	0	VCADD — 64-bit SIMD vector	Armv8.3
x1	0x	0	0	0	1	UNALLOCATED	-
x1	0x	0	0	1	0	VCADD — 128-bit SIMD vector	Armv8.3
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	Armv8.6
00	00	1	0	1	1	UNALLOCATED	-
00	00	1	1	0	0	VDOT (vector) — 64-bit SIMD vector	Armv8.6
00	00	1	1	0	1	UNALLOCATED	-
00	00	1	1	1	0	VDOT (vector) — 128-bit SIMD vector	Armv8.6
00	00	1	1	1	1	UNALLOCATED	-
00	01	1	0			UNALLOCATED	-
00	01	1	1			UNALLOCATED	-
00	10	0	0		1	VFMAL (vector)	Armv8.2
00	10	0	1			UNALLOCATED	-
00	10	1	0	0		UNALLOCATED	-
00	10	1	0	1	0	VSMMLA	Armv8.6
00	10	1	0	1	1	VUMMLA	Armv8.6
00	10	1	1	0	0	VSDOT (vector) — 64-bit SIMD vector	Armv8.2
00	10	1	1	0	1	VUDOT (vector) — 64-bit SIMD vector	Armv8.2
00	10	1	1	1	0	VSDOT (vector) — 128-bit SIMD vector	Armv8.2

Decode fields						Instruction Details	Architecture Version
op1	op2	op3	op4	Q	U		
00	10	1	1	1	1	VUDOT (vector) — 128-bit SIMD vector	Armv8.2
00	11	0	0		1	VFMAB, VFMAT (BFloat16, vector)	Armv8.6
00	11	0	1			UNALLOCATED	-
00	11	1	0			UNALLOCATED	-
00	11	1	1			UNALLOCATED	-
01	10	0	0		1	VFMSL (vector)	Armv8.2
01	10	0	1			UNALLOCATED	-
01	10	1	0	0		UNALLOCATED	-
01	10	1	0	1	0	VUSMMLA	Armv8.6
01	10	1	0	1	1	UNALLOCATED	-
01	10	1	1	0	0	VUSDOT (vector) — 64-bit SIMD vector	Armv8.6
01	10	1	1		1	UNALLOCATED	-
01	10	1	1	1	0	VUSDOT (vector) — 128-bit SIMD vector	Armv8.6
01	11	0	1			UNALLOCATED	-
01	11	1	0			UNALLOCATED	-
01	11	1	1			UNALLOCATED	-
	1x	0	0		0	VCMLA	Armv8.3
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 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
op	
0	VMAXNM
1	VMINNM

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	Architecture Version
01	UNALLOCATED	-
10	0	VMOVX
10	1	VINS
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	0	1	RM		Vd				1	0	!= 00	op	1	M	0		Vm		
																size															

The following constraints also apply to this encoding: size != 00 && size != 00

Decode fields o1 RM size op	Instruction Details
0	UNALLOCATED
0	VRINTA (floating-point)
0	VRINTN (floating-point)
0	VRINTP (floating-point)
0	VRINTM (floating-point)
1	VCVTA (floating-point)
1	VCVTN (floating-point)
1	VCVTP (floating-point)
1	VCVTM (floating-point)

Decode fields o1 RM	Instruction Details
0	VRINTA (floating-point)
0	VRINTN (floating-point)
0	VRINTP (floating-point)
0	VRINTM (floating-point)
1	VCVTA (floating-point)
1	VCVTN (floating-point)
1	VCVTP (floating-point)
1	VCVTM (floating-point)

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 op1 op2 Q U	Instruction Details	Architecture Version
0	VCMLA (by element) — 128-bit SIMD vector of half-precision floating-point	Armv8.3
0	00	VFMAL (by scalar)
0	01	VFMSL (by scalar)
0	10	UNALLOCATED

Decode fields				Instruction Details	Architecture Version
op1	op2	Q	U		
0	11		1	VFMAB, VFMAT (BFloat16, by scalar)	Armv8.6
1		0	0	VCMLA (by element) — 64-bit SIMD vector of single-precision floating-point	Armv8.3
1			1	UNALLOCATED	-
1		1	0	VCMLA (by element) — 128-bit SIMD vector of single-precision floating-point	Armv8.3

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	Architecture Version
op1	op2	op4	Q	U		
0	00	0			UNALLOCATED	-
0	00	1	0	0	VDOT (by element) — 64-bit SIMD vector	Armv8.6
0	00	1		1	UNALLOCATED	-
0	00	1	1	0	VDOT (by element) — 128-bit SIMD vector	Armv8.6
0	01	0			UNALLOCATED	-
0	10	0			UNALLOCATED	-
0	10	1	0	0	VSDOT (by element) — 64-bit SIMD vector	Armv8.2
0	10	1	0	1	VUDOT (by element) — 64-bit SIMD vector	Armv8.2
0	10	1	1	0	VSDOT (by element) — 128-bit SIMD vector	Armv8.2
0	10	1	1	1	VUDOT (by element) — 128-bit SIMD vector	Armv8.2
0	11				UNALLOCATED	-
1		0			UNALLOCATED	-
1	00	1	0	0	VUSDOT (by element) — 64-bit SIMD vector	Armv8.6
1	00	1	0	1	VSUDOT (by element) — 64-bit SIMD vector	Armv8.6
1	00	1	1	0	VUSDOT (by element) — 128-bit SIMD vector	Armv8.6
1	00	1	1	1	VSUDOT (by element) — 128-bit SIMD vector	Armv8.6
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

Decode fields	Instruction Details	
opc	L	
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

op0	Decode fields		op3	Instruction details
	op1	op2		
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

Decode fields		Instruction Details
L	sz	
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			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)

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 L	Instruction Details
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														
Rn																				Rt2				imm8							

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields L	Instruction Details
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	Decode fields imm3:imm2:stype	Rd	Instruction Details
0000	0				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					BIC, BICS (register)
0010	0	!= 1111			ORR, ORRS (register) — ORR
0010	0	1111			MOV, MOVS (register) — MOV
0010	1	!= 1111			ORR, ORRS (register) — ORRS
0010	1	1111			MOV, MOVS (register) — MOVS
0011	0	!= 1111			ORN, ORNS (register) — not flag setting
0011	0	1111			MVN, MVNS (register) — MVN
0011	1	!= 1111			ORN, ORNS (register) — flag setting
0011	1	1111			MVN, MVNS (register) — MVNS
0100	0				EOR, EORS (register) — EOR, rotate right with extend
0100	1		!= 0000011	!= 1111	EOR, EORS (register) — EORS, shift or rotate by value

Decode fields					Instruction Details
op1	S	Rn	imm3:imm2:type	Rd	
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			ADD, ADDS (register) — ADD
1000	0	1101			ADD, ADDS (SP plus register) — ADD
1000	1	!= 1101		!= 1111	ADD, ADDS (register) — ADDS
1000	1	1101		!= 1111	ADD, ADDS (SP plus register) — ADDS
1000	1			1111	CMN (register)
1001					UNALLOCATED
1010					ADC, ADCS (register)
1011					SBC, SBCS (register)
1100					UNALLOCATED
1101	0	!= 1101			SUB, SUBS (register) — SUB
1101	0	1101			SUB, SUBS (SP minus register) — SUB
1101	1	!= 1101		!= 1111	SUB, SUBS (register) — SUBS
1101	1	1101		!= 1111	SUB, SUBS (SP minus register) — SUBS
1101	1			1111	CMP (register)
1110					RSB, RSBS (register)
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
1	1	1	1	0	op0			op1		op2					1		op3			op4				op5							

Decode fields						Instruction details
op0	op1	op2	op3	op4	op5	
0	1110	0x	0x0		0	MSR (register)
0	1110	0x	0x0		1	MSR (Banked register)
0	1110	10	0x0	000		Hints
0	1110	10	0x0	!= 000		Change processor state
0	1110	11	0x0			Miscellaneous system
0	1111	00	0x0			BXJ
0	1111	01	0x0			Exception return

0	1111	1x	0x0		0	MRS
0	1111	1x	0x0		1	MRS (Banked register)
1	1110	00	000			DCPS
1	1110	00	010			UNALLOCATED
1	1110	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 hint	option	Instruction Details	Architecture Version
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	Armv8.2
0001	0001	Reserved hint, behaves as NOP	-
0001	0010	TSB CSYNC	Armv8.4
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 CPS
01		UNALLOCATED
10		CPS, CPSID, CPSIE — interrupt enable and change mode CPSIE
11		CPS, CPSID, CPSIE — interrupt disable and change mode CPSID

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
opc	option	
000x		UNALLOCATED
0010		CLREX
0011		UNALLOCATED
0100	!= 0x00	DSB
0100	0000	SSBB
0100	0100	PSSBB
0101		DMB
0110		ISB
0111		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	
	!= 00000000	SUB, SUBS (immediate)
1110	00000000	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		opt	Instruction Details
imm4	imm10		
!= 1111			UNALLOCATED
1111	!= 00000000000		UNALLOCATED
1111	00000000000	00	UNALLOCATED
1111	00000000000	01	DCPS1
1111	00000000000	10	DCPS2

Decode fields		Instruction Details	
imm4	imm10	opt	
1111	0000000000	11	DCPS3

Exception generation

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	1	1	o1					imm4	1	0	o2	0											imm12

Decode fields		Instruction Details
o1	o2	
0	0	HVC
0	1	UNALLOCATED
1	0	SMC
1	1	UDF

Data-processing (modified 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
1	1	1	1	0	i	0	op1			S	Rn			0	imm3	Rd			imm8												

Decode fields				Instruction Details
op1	S	Rn	Rd	
0000	0			AND, ANDS (immediate) — AND
0000	1		!= 1111	AND, ANDS (immediate) — ANDS
0000	1		1111	TST (immediate)
0001				BIC, BICS (immediate)
0010	0	!= 1111		ORR, ORRS (immediate) — ORR
0010	0	1111		MOV, MOVS (immediate) — MOV
0010	1	!= 1111		ORR, ORRS (immediate) — ORRS
0010	1	1111		MOV, MOVS (immediate) — MOVS
0011	0	!= 1111		ORN, ORNS (immediate) — not flag setting
0011	0	1111		MVN, MVNS (immediate) — MVN
0011	1	!= 1111		ORN, ORNS (immediate) — flag setting
0011	1	1111		MVN, MVNS (immediate) — MVNS
0100	0			EOR, EORS (immediate) — EOR
0100	1		!= 1111	EOR, EORS (immediate) — EORS
0100	1		1111	TEQ (immediate)
0101				UNALLOCATED
011x				UNALLOCATED
1000	0	!= 1101		ADD, ADDS (immediate) — ADD
1000	0	1101		ADD, ADDS (SP plus immediate) — ADD
1000	1	!= 1101	!= 1111	ADD, ADDS (immediate) — ADDS
1000	1	1101	!= 1111	ADD, ADDS (SP plus immediate) — ADDS
1000	1		1111	CMN (immediate)
1001				UNALLOCATED
1010				ADC, ADCS (immediate)
1011				SBC, SBCS (immediate)

Decode fields				Instruction Details
op1	S	Rn	Rd	
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
1	1	1	1	0		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							

Decode fields		Instruction Details
o1		
0		MOV, MOVS (immediate)
1		MOVT

Saturate, Bitfield

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	(0)	1	1	op1	0		Rn	0	imm3		Rd	imm2	(0)	widthm1													

Decode fields			Instruction Details
op1	Rn	imm3:imm2	
000			SSAT — logical shift left
001		!= 00000	SSAT — arithmetic shift right
001		00000	SSAT16
010			SBFX
011	!= 1111		BFI
011	1111		BFC
100			USAT — logical shift left
101		!= 00000	USAT — arithmetic shift right
101		00000	USAT16
110			UBFX
111			UNALLOCATED

Advanced SIMD element or structure load/store

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
								op0			0																				

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](#).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	0	0	1	0	D	L	0	Rn			Vd			itype			size		align		Rm						

Decode fields		Instruction Details
L	itype	
0	000x	VST4 (multiple 4-element structures)
0	0010	VST1 (multiple single elements) — T4
0	0011	VST2 (multiple 2-element structures) — T2
0	010x	VST3 (multiple 3-element structures)
0	0110	VST1 (multiple single elements) — T3
0	0111	VST1 (multiple single elements) — T1
0	100x	VST2 (multiple 2-element structures) — T1
0	1010	VST1 (multiple single elements) — T2
1	000x	VLD4 (multiple 4-element structures)
1	0010	VLD1 (multiple single elements) — T4
1	0011	VLD2 (multiple 2-element structures) — T2

Decode fields		Instruction Details
L	itype	
1	010x	VLD3 (multiple 3-element structures)
	1011	UNALLOCATED
1	0110	VLD1 (multiple single elements) — T3
1	0111	VLD1 (multiple single elements) — T1
	11xx	UNALLOCATED
1	100x	VLD2 (multiple 2-element structures) — T1
1	1010	VLD1 (multiple single elements) — T2

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	
0			UNALLOCATED
1	00		VLD1 (single element to all lanes)
1	01		VLD2 (single 2-element structure to all lanes)
1	10	0	VLD3 (single 3-element structure to all lanes)
1	10	1	UNALLOCATED
1	11		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	
0	00	00	VST1 (single element from one lane) — T1
0	00	01	VST2 (single 2-element structure from one lane) — T1
0	00	10	VST3 (single 3-element structure from one lane) — T1
0	00	11	VST4 (single 4-element structure from one lane) — T1
0	01	00	VST1 (single element from one lane) — T2
0	01	01	VST2 (single 2-element structure from one lane) — T2
0	01	10	VST3 (single 3-element structure from one lane) — T2
0	01	11	VST4 (single 4-element structure from one lane) — T2
0	10	00	VST1 (single element from one lane) — T3
0	10	01	VST2 (single 2-element structure from one lane) — T3
0	10	10	VST3 (single 3-element structure from one lane) — T3
0	10	11	VST4 (single 4-element structure from one lane) — T3
1	00	00	VLD1 (single element to one lane) — T1
1	00	01	VLD2 (single 2-element structure to one lane) — T1
1	00	10	VLD3 (single 3-element structure to one lane) — T1

Decode fields			Instruction Details
L	size	N	
1	00	11	VLD4 (single 4-element structure to one lane) — T1
1	01	00	VLD1 (single element to one lane) — T2
1	01	01	VLD2 (single 2-element structure to one lane) — T2
1	01	10	VLD3 (single 3-element structure to one lane) — T2
1	01	11	VLD4 (single 4-element structure to one lane) — T2
1	10	00	VLD1 (single element to one lane) — T3
1	10	01	VLD2 (single 2-element structure to one lane) — T3
1	10	10	VLD3 (single 3-element structure to one lane) — T3
1	10	11	VLD4 (single 4-element structure to one lane) — T3

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)

Decode fields			Instruction Details
size	L	Rt	
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)
11			UNALLOCATED

Load/store, unsigned (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	0	0	size	L	!= 1111	Rt				1	1	1	0	imm8											
Rn																															

The following constraints also apply to this encoding: Rn != 1111 && Rn != 1111

Decode fields		Instruction Details
size	L	
00	0	STRBT
00	1	LDRBT
01	0	STRHT
01	1	LDRHT
10	0	STRT
10	1	LDRT
11		UNALLOCATED

Load/store, unsigned (immediate, pre-indexed)

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	U	1												
Rn																imm8															

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 (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	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	1	!= 1111		Rt		1	0	U	1												

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	1	!= 1111		Rt		1	1	0	0												

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	1	!= 1111		Rt		1	1	1	0												

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	1	!= 1111		Rt		1	1	U	1												

Rn

The following constraints also apply to this encoding: $Rn \neq 1111$ && $Rn \neq 1111$

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	1	!= 1111		Rt																	
Rn																imm12															

The following constraints also apply to this encoding: $Rn \neq 1111$ && $Rn \neq 1111$

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			Instruction details
op0	op1	op2	
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

Decode fields				Instruction Details
op1	U	H	S	
010	0	0	0	SASX
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

Decode fields		Instruction Details
op1	op2	
001	10	RBIT
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

Decode fields			Instruction Details
op1	Ra	op2	
001	!= 1111	11	SMLABB, SMLABT, SMLATB, SMLATT — SMLATT
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, SMLSDX — SMLSD
100	!= 1111	01	SMLSD, SMLSDX — SMLSDX
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

Decode fields		Instruction Details
op1	op2	
011	1111	UDIV
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 LDx — SMLS LD
101	1101	SMLS LD, SMLS LDx — SMLS LDx
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_19v01-15, pseudocode v2020-09_xmlv2020-06-rel, sve v2020-09_rc3v2020-06-29-ge9614a3 ; Build timestamp: 2020-09-30T21:2020-07-03T11:3536

Copyright © 2010-2020 Arm Limited or its affiliates. All rights reserved. This document is Non-Confidential.

(old)

htmldiff from-

(new)